Salesforce Trailhead : Developer : Intermediate

Get to Know the Salesforce APIs

Learning Objectives

After completing this unit, you'll be able to:

- Describe the benefits of the API-first approach to development.
- State use cases for REST API, SOAP API, Bulk API, and Streaming API.
- Name the two types of API limits and describe how they're calculated.

API First at Salesforce

Ahoy, matey! Are you ready to set sail on the high seas in search of the perfect Salesforce API for the integration you're building? Well, captain, grab your eye patch and find a parrot. You're about to navigate the Force.com waters and learn all about our APIs.

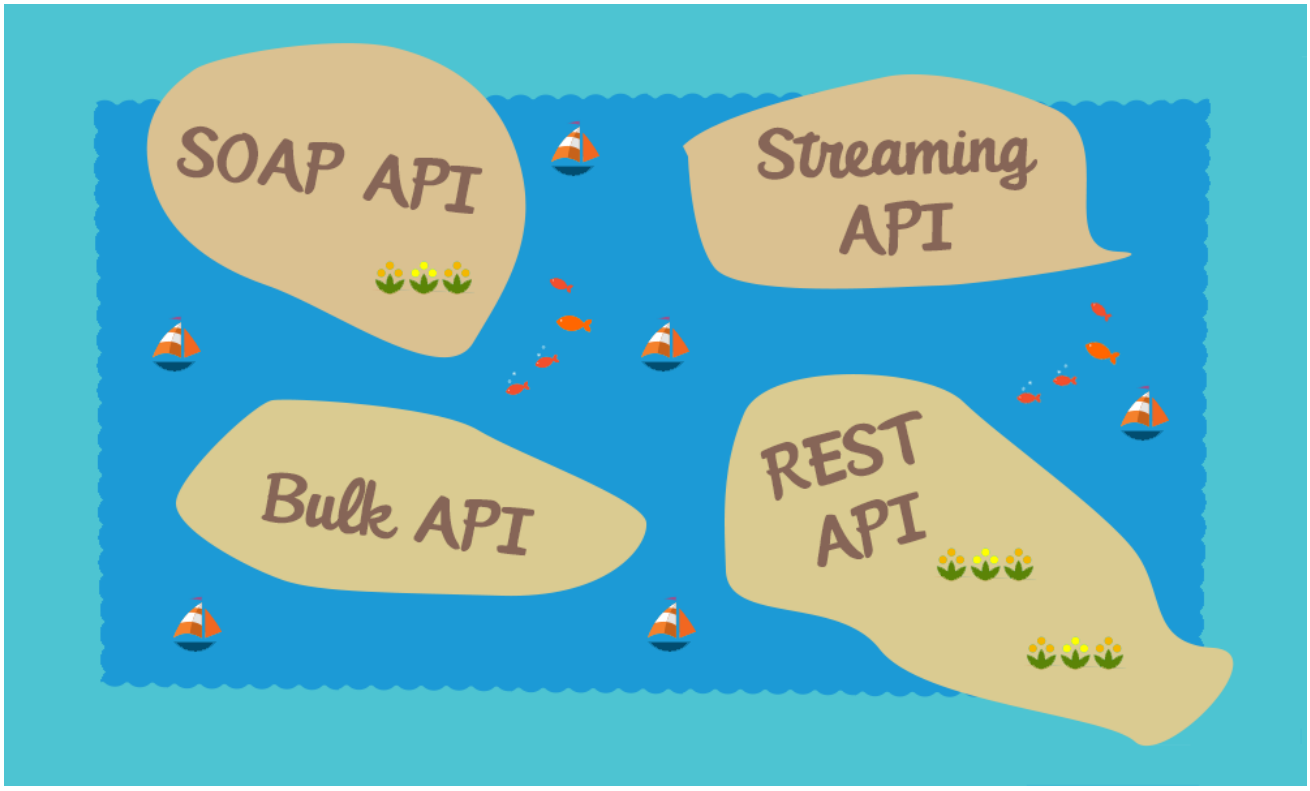
The Salesforce API landscape is as vast as the ocean blue. That's because Salesforce takes an API-first approach to building features on the Salesforce App Cloud. API first means building a robust API for a feature before focusing on designing its UI. This approach gives you, the Salesforce developer, flexibility to manipulate your data however you want.

Salesforce knows that its customers and partners are always thinking of new ways to extend Salesforce functionality and exciting apps to build for the AppExchange. Providing a comprehensive toolbox for developing on the platform is a top priority. This approach also lets Salesforce build UIs on top of the APIs, ensuring that the behavior is the same between them.

Think of this module as your API first mate. Together we'll walk through some general API info, perform a survey of Salesforce's API suite, and dive into using a few common APIs. All this information will equip you with the knowledge you need to choose the right API for your project.

Salesforce Data APIs

In the sea of Salesforce APIs, there's a key archipelago of commonly used APIs that we'll focus on in this module. They are REST API, SOAP API, Bulk API, and Streaming API. Together they make up the Salesforce data APIs. Their purpose is to let you manipulate your Salesforce data, whereas other APIs let you do things like customize page layouts or build custom development tools. You can use other Salesforce APIs to manipulate subsets of your Salesforce data, too. For example, Wave REST API focuses on Wave Analytics. But these four APIs apply broadly across the spectrum of core Salesforce data.



REST API

REST API is a simple and powerful web service based on RESTful principles. It exposes all sorts of Salesforce functionality via REST resources and HTTP methods. For example, you can create, read, update, and delete (CRUD) records, search or query your data, retrieve object metadata, and access information about limits in your org. REST API supports both XML and JSON.

Because REST API has a lightweight request and response framework and is easy to use, it's great for writing mobile and web apps.

SOAP API

SOAP API is a robust and powerful web service based on the industry-standard protocol of the same name. It uses a Web Services Description Language (WSDL) file to rigorously define the parameters for accessing data through the API. SOAP API supports XML only. Most of the SOAP API functionality is also available through REST API. It just depends on which standard better meets your needs.

Because SOAP API uses the WSDL file as a formal contract between the API and consumer, it's great for writing server-to-server integrations.

Bulk API

Bulk API is a specialized RESTful API for loading and querying lots of data at once. By lots, we mean 50,000 records or more. Bulk API is asynchronous, meaning that you can submit a request and come back later for the results. This approach is the preferred one when dealing with large amounts of data.

Bulk API is great for performing tasks that involve lots of records, such as loading data into your org for the first time.

Streaming API

Streaming API is a specialized API for setting up notifications that trigger when changes are made to your data. It uses a publish-subscribe, or pub/sub, model in which users can subscribe to channels that broadcast certain types of data changes.

The pub/sub model reduces the number of API requests by eliminating the need for polling. Streaming API is great for writing apps that would otherwise need to frequently poll for changes.

API Access and Authentication

You don't need a treasure map to access Salesforce APIs. All you need is an org on one of the following editions: Enterprise Edition, Unlimited Edition, Developer Edition, Performance Edition, or Professional Edition (with an add-on). Make sure that you have the "API Enabled" permission, and you're ready to start integrating.

All API calls, except for the SOAP API `login()` call, require authentication. You can either use one of the supported OAuth flows or authenticate with a session ID retrieved from the SOAP API `login()` call. Check the developer guide for your API of choice to get started.

API Limits

Any captain worth his or her salt knows when to put limits on the crew for the good of the ship. If the captain lets the sailors drink grog all day long, nothing gets done. Similarly, Salesforce limits the number of API calls per org to ensure the health of the instance. These limits exist to prevent rogue scripts from smashing our servers into driftwood. They don't get in the way of your everyday work. Nonetheless, it's a good idea to get familiar with them.

There are two types of API limits. Concurrent limits cap the number of long-running calls (20 seconds or longer) that are running at one time. Total limits cap the number of calls made within a rolling 24-hour period.

Concurrent limits vary by org type. For a Developer Edition org, the limit is five long-running calls at once. For a sandbox org, it's 25 long-running calls.

Total limits vary by org edition, license type, and expansion packs that you purchase. For example, an Enterprise Edition org gets 1,000 calls per Salesforce license and 200 calls per Force.com Light App license. With the Unlimited Apps Pack, that same Enterprise Edition org gets an extra 4,000 calls. Total limits are also subject to minimums and maximums based on the org edition, but we won't get into that here. If you want to know more, check out the API Request Limits link in the Resources section.

You have several ways to check your remaining API calls. You can view them in the API Usage box on the System Overview page. From Setup, enter `System Overview` in the Quick Find box, then select **System Overview**. You can also set up notifications for when your org exceeds a number of API calls that you designate. To do so, from Setup, enter `API Usage Notifications` in the Quick Find box, then select **API Usage Notifications**.

When using REST or SOAP API, the LimitInfoHeader response header gives you information on your remaining calls. You can also access the REST API Limits resource for information about all sorts of limits in your org.

Which API Do I Use?

Choosing the right API for your integration needs is an important decision. Here's some information on our most commonly used APIs, including supported protocols, data formats, communication paradigms, and use cases. Treat this section as a reference you can return to when you're considering which API to use.

Note the four data APIs that we talked about already. We'll be diving into each of them next.

API Name	Protocol	Data Format	Communication
REST API	REST	JSON, XML	Synchronous
SOAP API	SOAP (WSDL)	XML	Synchronous
Chatter REST API	REST	JSON, XML	Synchronous (photos are processed asynchronously)
Wave Analytics REST API	REST	JSON, XML	Synchronous
Bulk API	REST	CSV, JSON, XML	Asynchronous
Metadata API	SOAP (WSDL)	XML	Asynchronous
Streaming API	Bayeux	JSON	Asynchronous (stream of data)
Apex REST API	REST	JSON, XML, Custom	Synchronous
Apex SOAP API	SOAP (WSDL)	XML	Synchronous
Tooling API	REST or SOAP (WSDL)	JSON, XML, Custom	Synchronous

When to Use REST API

REST API provides a powerful, convenient, and simple REST-based web services interface for interacting with Salesforce. Its advantages include ease of integration and development, and it's an excellent choice of technology for use with mobile applications and web projects. However, if you have many records to process, consider using Bulk API, which is based on REST principles and optimized for large sets of data.

When to Use SOAP API

SOAP API provides a powerful, convenient, and simple SOAP-based web services interface for interacting with Salesforce. You can use SOAP API to create, retrieve, update, or delete records. You can also use SOAP API to perform searches and much more. Use SOAP API in any language that supports web services.

For example, you can use SOAP API to integrate Salesforce with your org's ERP and finance systems. You can also deliver real-time sales and support information to company portals and populate critical business systems with customer information.

When to Use Chatter REST API

Use Chatter REST API to display Salesforce data, especially in mobile applications. In addition to Chatter feeds, users, groups, and followers, Chatter REST API provides programmatic access to files, recommendations, topics, notifications, Data.com purchasing, and more. Chatter REST API is similar to APIs offered by other companies with feeds, such as Facebook and Twitter, but it also exposes Salesforce features beyond Chatter.

When to Use the Wave Analytics REST API

You can access Wave Analytics assets—such as datasets, lenses, and dashboards—programmatically using the Wave REST API. Send queries directly to the Wave Platform. Access datasets that have been imported into the Wave Platform. Create and retrieve Wave Analytics lenses. Access XMD information. Retrieve a list of dataset versions. Create and retrieve Wave Analytics applications. Create, update, and retrieve Wave Analytics dashboards. Retrieve a list of dependencies for an application. Determine what features are available to the user. Work with snapshots. Manipulate replicated datasets.

When to Use Bulk API

Bulk API is based on REST principles and is optimized for loading or deleting large sets of data. You can use it to query, queryAll, insert, update, upsert, or delete many records asynchronously by submitting batches. Salesforce processes batches in the background.

SOAP API, in contrast, is optimized for real-time client applications that update a few records at a time. You can use SOAP API for processing many records, but when the data sets contain hundreds of thousands of records, SOAP API is less practical. Bulk API is designed to make it simple to process data from a few thousand to millions of records.

The easiest way to use Bulk API is to enable it for processing records in Data Loader using CSV files. Using Data Loader avoids the need to write your own client application.

When to Use Metadata API

Use Metadata API to retrieve, deploy, create, update, or delete customizations for your org. The most common use is to migrate changes from a sandbox or testing org to your production environment. Metadata API is intended for managing customizations and for building tools that can manage the metadata model, not the data itself.

The easiest way to access the functionality in Metadata API is to use the Force.com IDE or Force.com Migration Tool. Both tools are built on top of Metadata API and use the standard Eclipse and Ant tools, respectively, to simplify working with Metadata API.

- Force.com IDE is built on the Eclipse platform, for programmers familiar with integrated development environments. Code, compile, test, and deploy from within the IDE.
- The Force.com Migration Tool is ideal if you use a script or the command line for moving metadata between a local directory and a Salesforce org.

When to Use Streaming API

Use Streaming API to receive notifications for changes to data that match a SOQL query that you define.

Streaming API is useful when you want notifications to be pushed from the server to the client. Consider Streaming API for applications that poll frequently. Applications that have constant polling against the Salesforce infrastructure consume unnecessary API call and processing time. Streaming API reduces the number of requests that return no data, and is also ideal for applications that require general notification of data changes.

Streaming API enables you to reduce the number of API calls and improve performance.

When to Use Apex REST API

Use Apex REST API when you want to expose your Apex classes and methods so that external applications can access your code through REST architecture. Apex REST API supports both OAuth 2.0 and Session ID for authorization.

When to Use Apex SOAP API

Use Apex SOAP API when you want to expose Apex methods as SOAP web service APIs so that external applications can access your code through SOAP.

Apex SOAP API supports both OAuth 2.0 and Session ID for authorization.

When to Use Tooling API

Use Tooling API to integrate Salesforce metadata with other systems. Metadata types are exposed as sObjects, so you can access one component of a complex type. This field-level access speeds up operations on complex metadata types. You can also build custom development tools for Force.com applications. For example, use Tooling API to manage and deploy working copies of Apex classes and triggers and Visualforce pages and components. You can also set checkpoints or heap dump markers, execute anonymous Apex, and access logging and code coverage information.

REST and SOAP are both supported.

Resources

Use REST API

Learning Objectives

After completing this unit, you'll be able to:

- Log in to Workbench and navigate to REST Explorer.
- Use the describe resource.
- Create an account using REST API.
- Execute a query using REST API.

REST Resources and Methods

Land ho! We've spotted the Isle of REST ahead of the bow, captain. Before we dock and start using the API, let's talk about REST resources and methods.

A REST resource is an abstraction of a piece of information or an action, such as a single data record, a collection of records, or a query. Each resource in REST API is identified by a named Uniform Resource Identifier (URI) and is accessed using standard HTTP methods (HEAD, GET, POST, PATCH, DELETE). REST API is based on the usage of resources, their URIs, and the links between them.

You use a resource to interact with your Salesforce org. For example, you can:

- Retrieve summary information about the API versions available to you.
- Obtain detailed information about a Salesforce object, such as Account, User, or a custom object.
- Perform a query or search.
- Update or delete records.

A REST request consists of four components: a resource URI, an HTTP method, request headers, and a request body. Request headers specify metadata for the request. The request body specifies data for the request, when necessary. If there's no data to specify, the body is omitted from the request.

Describe the Account Object

It's time to get our feet wet. We're going to use Workbench to make some API calls. Workbench is a suite of tools for interacting with your Salesforce org through the API. Because you can make REST requests from any HTTP sender, there are plenty of other tools available for you to use (for example, check out cURL or Postman). But because Workbench provides a friendly framework specifically for Salesforce APIs, it's the perfect way to dig in before you're ready to write a full-on integration.
The first step is to log in to Workbench.

1. Log in to your Trailhead DE org and navigate to Workbench.
2. For Environment, select **Production**.
3. For API Version, select the highest available number.
4. Make sure that you select **I agree to the terms of service**.
5. Click **Login with Salesforce**.

You've arrived at the Workbench home page. For this module, we use only one of Workbench's many tools, the REST Explorer.

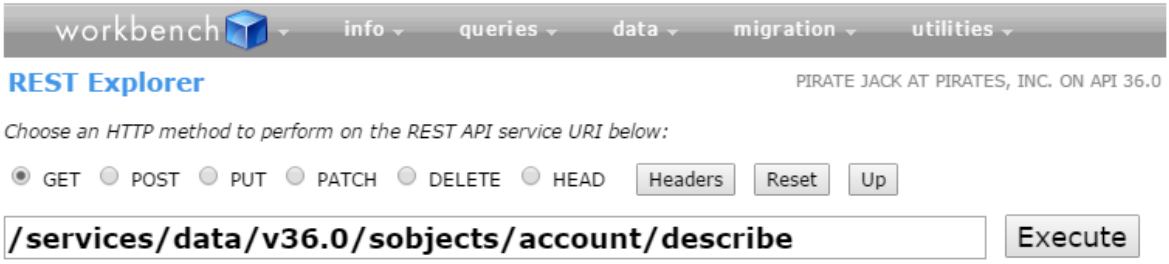
In the top menu, select .

You can make REST API calls from the REST explorer just like you would from any other HTTP interface. The text in the text box represents a resource URI. For convenience, the top-level domain is omitted from the displayed URI. For example, the full URI of the resource that's prepopulated in the URI text box is https://foo.my.salesforce.com/services/data/v36.0.

The radio buttons above the URI represent the standard HTTP methods. To make an API call, enter the resource URI, select the appropriate method, add headers as needed, and click **Execute**.

Let's try out the SObject Describe resource. This resource, when combined with the GET method, returns metadata about an object and its fields.

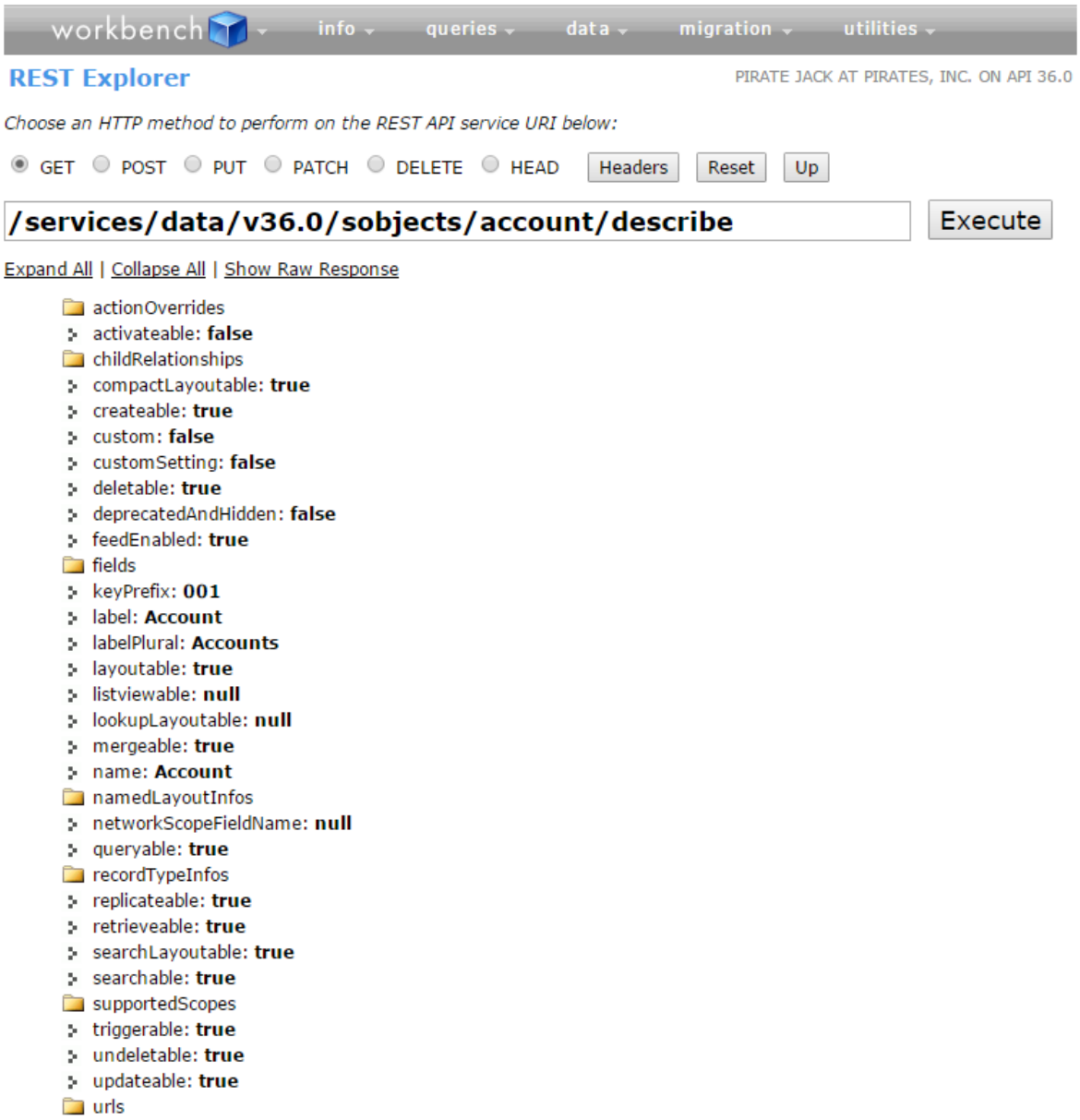
We'll try describing the Account object. Replace the existing text in the URI text box with /services/data/v*XX*.0/sobjects/account/describe, where *XX* maps to the API version you're using.



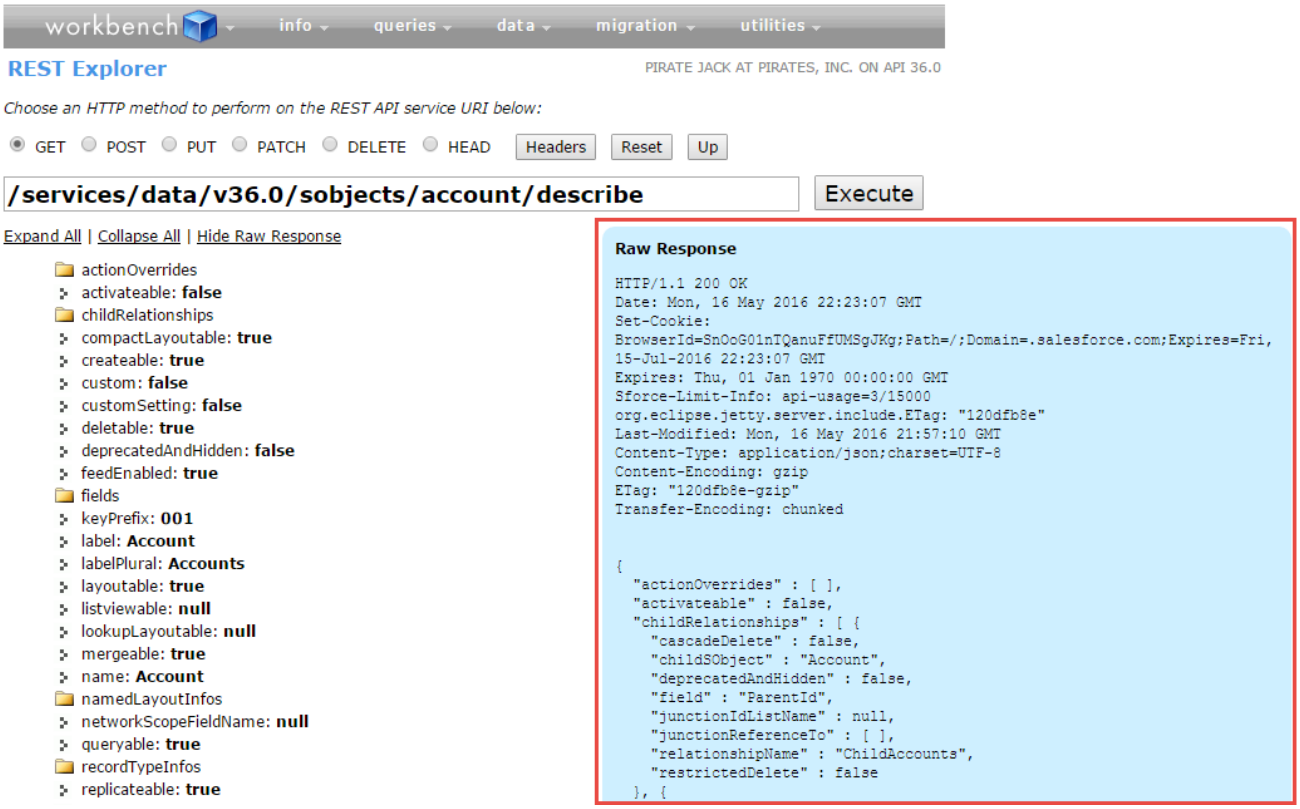
Let's take a minute to break down this resource's URI.

- `/services/data`—Specifies that we're making a REST API request
- `/v36.0`—API version number
- `/sobjects`—Specifies that we're accessing a resource under the sObject grouping
- `/account`—sObject being actioned; in this case, account
- `/describe`—Action; in this case, a describe request

Now make sure that the GET method is selected, and click **Execute**.



Good work, captain. The Account metadata appears on the screen. And Workbench has nicely formatted the response. To see the raw JSON response, click **Show Raw Response**.



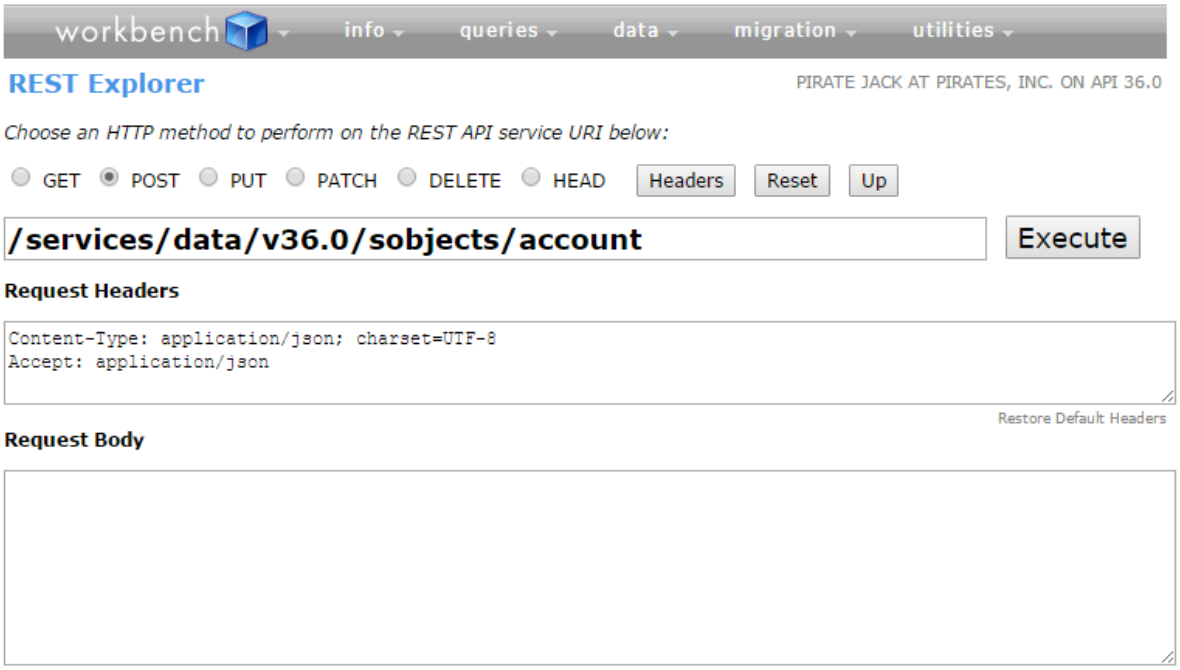
The Account metadata is displayed in JSON below some HTTP response headers. Because REST API supports both JSON and XML, let's change the request header to specify an XML response. Next to the HTTP methods, click **Headers**. For the `Accept` header value, replace `application/json` with `application/xml`. Your request headers look like this.

Click **Execute**. The raw XML response is returned. Hurrah!

Create an Account

Now let's create an account using the SObject resource and the POST method. In the URI text box, replace the existing text with /services/data/v*XX*.0/sobjects/account, where *XX* maps to the API version you're using. Select **POST**. Notice that a Request Body text area appears, which is where we specify the field values for our new account. First, though, let's change the `Accept` header back to JSON.

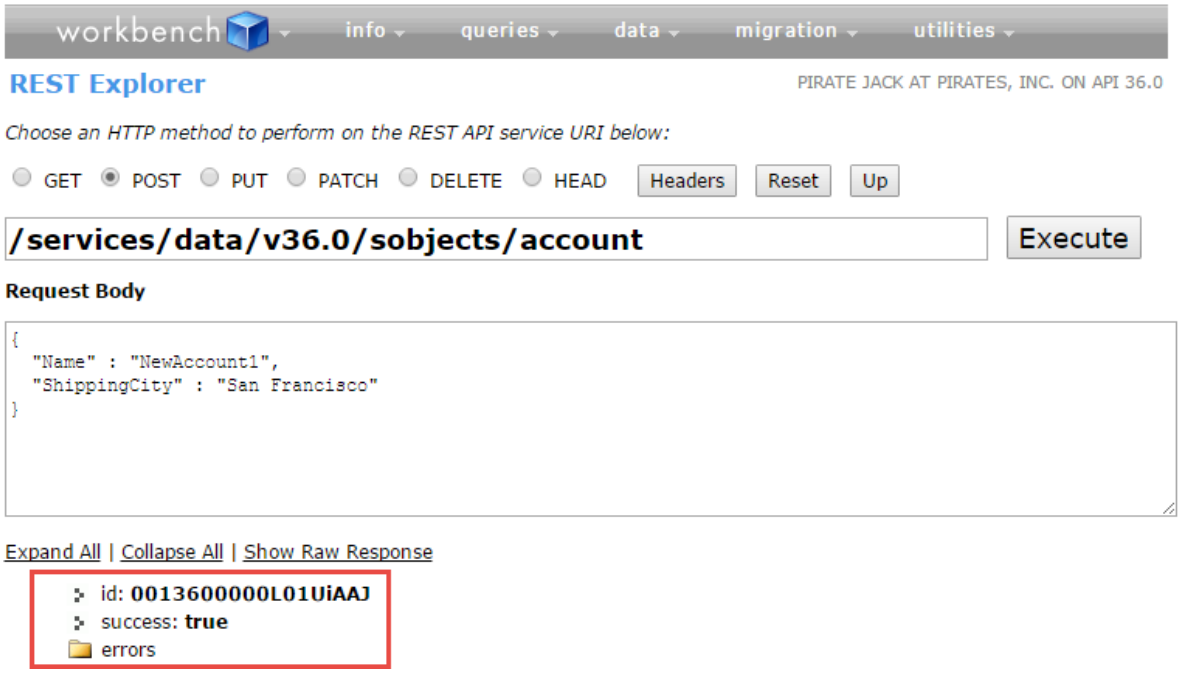
Click **Headers**. Change `Accept: application/xml` back to `Accept: application/json`. Your request looks like this.



In the request body, enter the following text.

```
{
  "Name" : "NewAccount1",
  "ShippingCity" : "San
Francisco"
}
```

Click **Execute**. You see a response such as the following.



If `success: true`, the account was created with the returned ID. Expand the errors folder to check for errors.

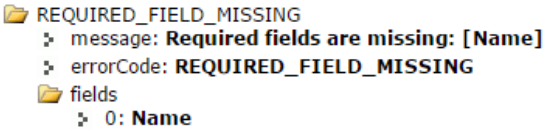
Just for kicks, let's create a second account without specifying an account name. Replace the text in the request body with the following text.

```
{
  "ShippingCity" : "San
Francisco"
}
```

Click **Execute**.

Uh, oh. Did you get a `REQUIRED_FIELD_MISSING` response? Expand the **REQUIRED_FIELD_MISSING** folder, and then expand the **fields** folder. Your expanded response looks like this.

Because Name is a required field for creating an account, the server didn't process the request, and we received an error. Thankfully, all the information we need to fix the request is in the error response. Let's specify the name `NewAccount2`, and change the shipping city in the request body. Replace the text in the request body with the following text.




```
{
  "Name" : "NewAccount2",
  "ShippingCity" : "New
York"
}
```

Click **Execute**. Success!

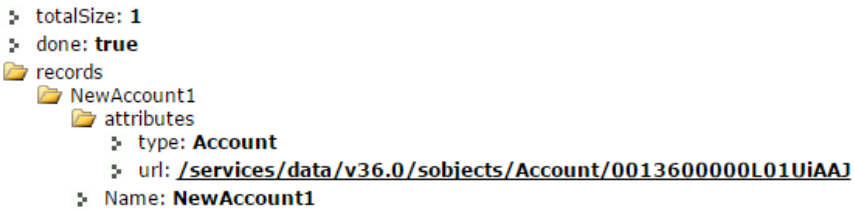
Execute a Query

Now let's imagine that you or another user has created hundreds of accounts. You want to find the names of all the accounts where the shipping city is San Francisco. You can use the Query resource to execute a SOQL query and zero in on the exact records you want, just like a customized treasure map.

Replace the text in the URI text box with the following text: `/services/data/vXX.0/query/?q=SELECT+Name+From+Account+WHERE+ShippingCity='San+Francisco'`, where *XX* maps to the API version you're using.

We replaced spaces with the + character in the query string to properly encode the URI. You can read about HTML URL encoding from the link in the Resources section. Make sure that the GET method is selected, and click **Execute**.

Expand the **records** folder. Do you see a folder with the name of the first account we created, NewAccount1? Great. Click it. Now click the **attributes** folder. Next to url is the resource URI of the account that was returned. Your response looks something like this.



When you write an integration, you can grab this URI from the response to access more details about that account.

Node.js and Ruby Samples

Now you have a sweet taste of what's possible with REST API. Of course, when you move from Workbench to writing code, you'll be interacting with REST API using the programming language of your choice. Luckily, expert Salesforce developers have written wrappers for several languages that simplify the process of consuming REST API. Here are two sample queries written in Node.js and Ruby that use wrappers Nforce and Restforce, respectively.

Node.js Sample Using Nforce

```
var nforce = require('nforce');

// create the connection with the Salesforce connected app
var org = nforce.createConnection({
  clientId: process.env.CLIENT_ID,
  clientSecret: process.env.CLIENT_SECRET,
  redirectUri: process.env.CALLBACK_URL,
  mode: 'single'
});

// authenticate and return OAuth token
org.authenticate({
  username: process.env.USERNAME,
  password: process.env.PASSWORD+process.env.SECURITY_TOKEN
}, function(err, resp){
  if (!err) {
    console.log('Successfully logged in! Cached Token: ' + org.oauth.access_token);
    // execute the query
    org.query({ query: 'select id, name from account limit 5' }, function(err,
resp){
      if(!err && resp.records) {
        // output the account names
        for (i=0; i<resp.records.length;i++) {
          console.log(resp.records[i].get('name'));
        }
      }
    });
  }
  if (err) console.log(err);
});
```

Ruby Sample Using Restforce

```
require 'restforce'

// create the connection with the Salesforce connected app
client = Restforce.new :username => ENV['USERNAME'],
  :password       => ENV['PASSWORD'],
  :security_token => ENV['SECURITY_TOKEN'],
  :client_id      => ENV['CLIENT_ID'],
  :client_secret  => ENV['CLIENT_SECRET']

// execute the query
accounts = client.query("select id, name from account limit
5")

// output the account names
accounts.each do |account|
  p account.Name
end
```

Resources

Use SOAP API

Learning Objectives

After completing this unit, you'll be able to:

- Generate a WSDL file for your org.
- Use SoapUI to create a SOAP project from the WSDL file.
- Log in to your DE org using SOAP API.
- Create an account using SOAP API.

Enterprise and Partner WSDLs

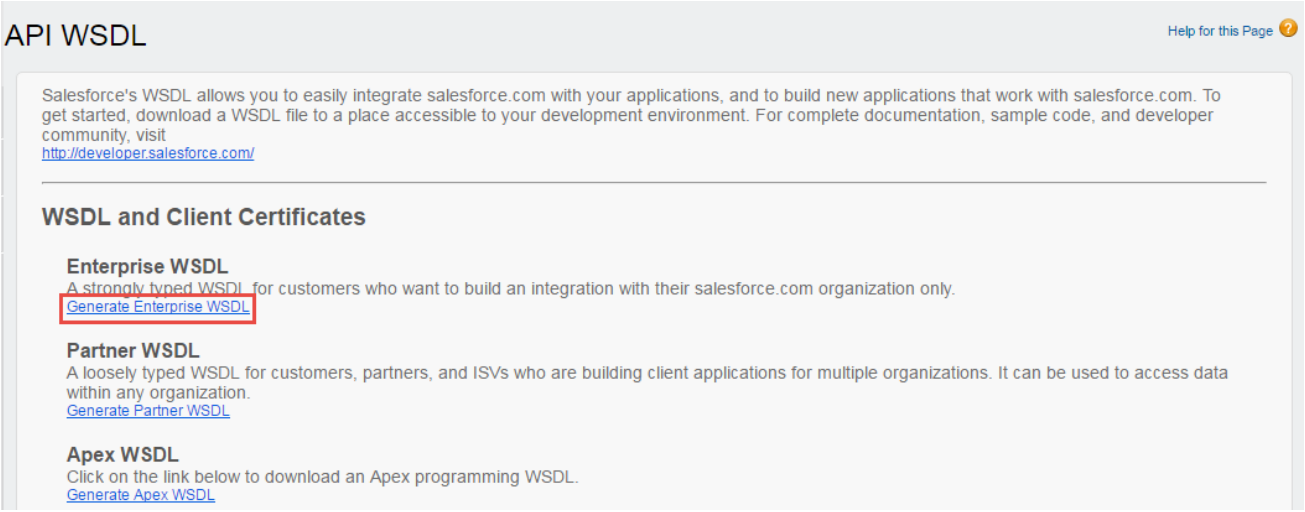
If you've sailed the straits of another SOAP-based API, you know that the Web Services Description Language (WSDL) file is basically your map to understanding how to use the API. It contains the bindings, protocols, and objects to make API calls.

Salesforce provides two SOAP API WSDLs for two different use cases. The enterprise WSDL is optimized for a single Salesforce org. It's strongly typed, and it reflects your org's specific configuration, meaning that two enterprise WSDL files generated from two different orgs contain different information.

The partner WSDL is optimized for use with many Salesforce orgs. It's loosely typed, and it doesn't change based on an org's specific configuration.

Typically, if you're writing an integration for a single Salesforce org, use the enterprise WSDL. For several orgs, use the partner WSDL.

For this unit, we're using the enterprise WSDL to explore SOAP API. The first step is to generate a WSDL file for your org. In your DE org, from Setup, enter `API` in the Quick Find box, then select **API**. On the API WSDL page, click **Generate Enterprise WSDL**.



On the Generate Enterprise WSDL page, click **Generate**. When the WSDL is generated, right-click on the page and save the WSDL file somewhere on your computer. We'll be using it shortly.

Here's a tip for working with the enterprise WSDL. The enterprise WSDL reflects your org's specific configuration, so whenever you make a metadata change to your org, regenerate the WSDL file. This way, the WSDL file doesn't fall out of sync with your org's configuration.

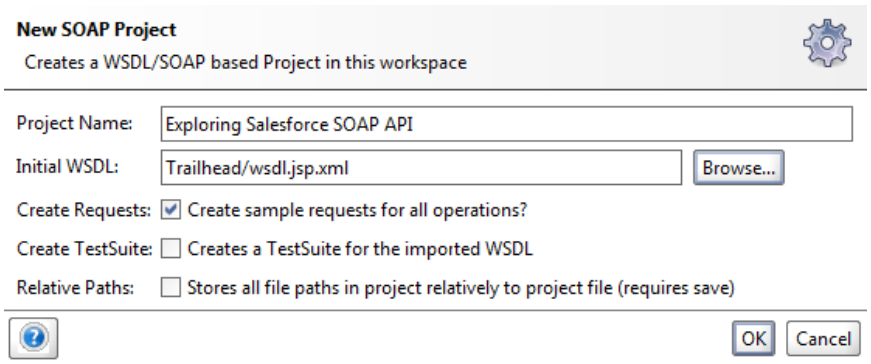
Create a SOAP Project with SoapUI

Now that we have our WSDL file, we need a way to extract the information to start making SOAP API requests. In web parlance, this process is called consuming the WSDL. Much like a kraken consumes a ship full of hapless sailors, tools such as Force.com Web Services Connector (WSC) consume the WSDL file. The tools then create classes that enable you to make requests with SOAP API using common programming languages.

For this unit, we're using a third-party tool called SoapUI to consume our enterprise WSDL file. SoapUI is a free and open-source app for testing web services. To get started, download and install SoapUI OpenSource from the SoapUI website. Install only the SoapUI component.

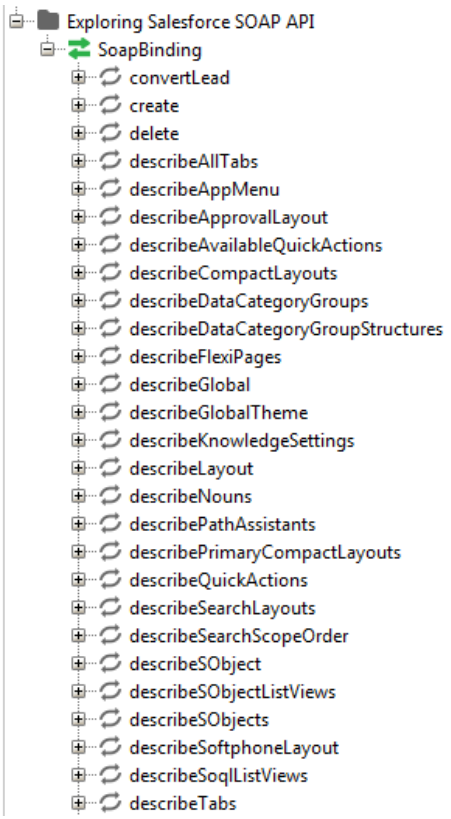
Here's a video that shows the process of consuming the enterprise WSDL and making SOAP API requests with SoapUI. To walk through it yourself, follow the instructions below the video.

After you get SoapUI installed and launched, from the File menu, select **New SOAP Project**. For the project name, enter `Exploring Salesforce SOAP API`. For the initial WSDL, browse to where you saved the enterprise WSDL file and select it. Don't change any other options. Your SoapUI window looks something like this.



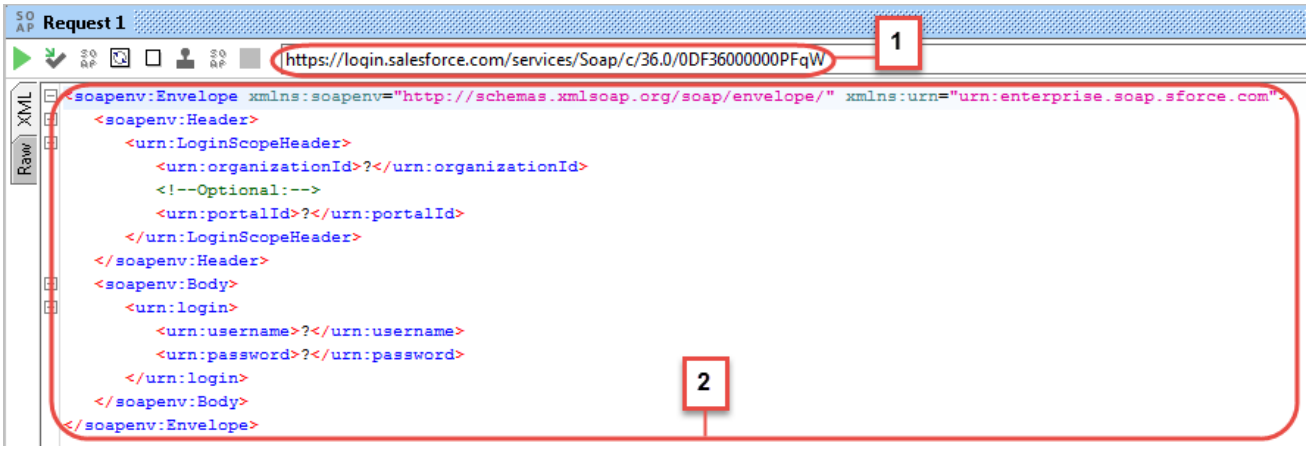
Click **OK**. After a few seconds of processing, an Exploring Salesforce SOAP API folder appears in the navigator pane on the left side of the screen. Underneath it is an entry called SoapBinding that contains several operations.

So what are we looking at here? Each operation corresponds to a SOAP API request we can make. The properties of each operation are pulled from information in the WSDL file. Each operation also contains a sample XML request that includes the operation's HTTPS endpoint and a prepopulated SOAP message. Now we're all set to make SOAP API requests.



Log In to Your DE Org

In SoapUI, scroll down to the `login` operation. Expand it, and then double-click **Request 1**. A sample SOAP login request appears.



Here's a quick breakdown of the endpoint URI (1).

- `https://`—Specifies secure HTTP.
- `login.salesforce.com`—Top-level domain for a login request.
- `/services/Soap`—Specifies that we're making a SOAP API request.
- `/c`—Specifies that we're using the enterprise WSDL. Use `/u` for the partner WSDL.
- `/36.0`—API version number. The `v` prefix is missing because some APIs include it before the version number, and some don't. This behavior is just a quirk of Salesforce APIs.
- `/0DF36000000LHZw`—Package version number.

We aren't using managed packages for this example, so we can remove the package version number from the end of the URI. Go ahead and do that now.

The SOAP message (2) contains everything we expect to find in a SOAP message: an envelope, a header, and a body.

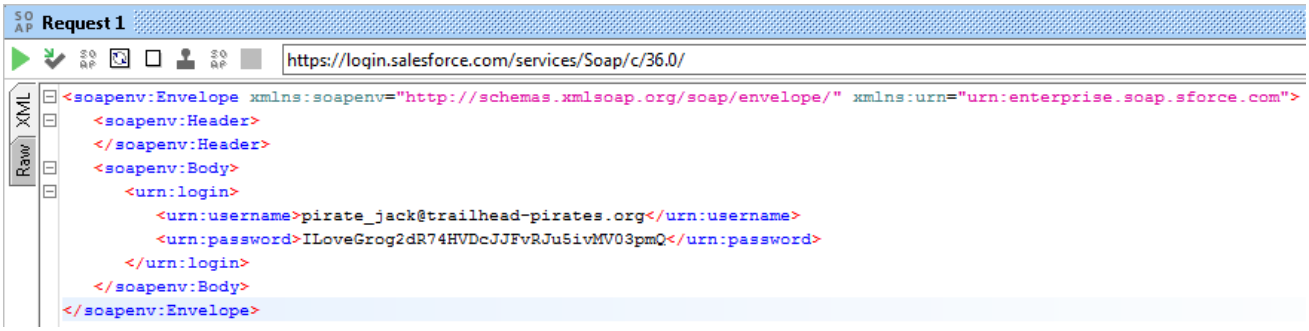
The properties inside the `LoginScopeHeader` element concern the authentication of Self-Service and Customer Portal users. Because we don't have to worry about these values for our purposes, delete the entire element (everything from to). Highlight the text in the window, and press the Delete key.

Next, look at the element in the message body. This element is the meat of the login request. It's where we provide our user credentials. Replace the ?s with your username and password for your DE org.

Since you're making an API request from an IP address unknown to Salesforce, you need to append your security token to the end of your password. For example, if your password is `mypassword` and your security token is `XXXXXXXXXX`, enter `mypasswordXXXXXXXXXX` inside the element. Your security token was emailed to you when you first reset your password after signing up for your DE org. If you've reset your password since then, a new security token was emailed to you. If you can't find it, you can reset it in Salesforce.

1. From your personal settings, enter `Reset` in the Quick Find box, then select **Reset My Security Token**.
2. Click **Reset Security Token**. The new security token is sent to the email address in your Salesforce personal settings.

Your SOAP message looks something like this.




```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:urn="urn:enterprise.soap.sforce.com">
   <soapenv:Header>
   </soapenv:Header>
   <soapenv:Body>
      <urn:login>
         <urn:username>pirate_jack@trailhead-pirates.org</urn:username>
         <urn:password>ILoveGrog2dR74HVDcJJFvRJu5ivMV03pmQ</urn:password>
      </urn:login>
   </soapenv:Body>
</soapenv:Envelope>
```

Click the play button (green triangle) in the upper left of the request window. This button sends the request, casting your SOAP message-in-a-bottle into the sea. If you get an error about needing to use TLS 1.1 or higher, see this blog post to fix it. Otherwise, check out the response. Here's what it looks like when we expand it.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns="urn:enterprise.soap.sforce.com"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <loginResponse>
         <result>
            <metadataServerUrl>https://na30.salesforce.com/services/Soap/m/36.0/00D36000000wCdk</metadataServerUrl>
            <passwordExpired>false</passwordExpired>
            <sandbox>false</sandbox>
            <serverUrl>https://na30.salesforce.com/services/Soap/c/36.0/00D36000000wCdk</serverUrl>
            <sessionId>00D36000000wCdk!AQEAQK23h7Ak_SyGTu_WrzC1a2KTGnZacSOWiR4jc8Mp2Jt7_f4t8XkJ.g64WlP7_JfweOHn.Ak.SwUEkQR0XGcKAxKq4gtU</sessionId>
            <userId>00536000002BU42AAG</userId>
            <userInfo>
               <accessibilityMode>false</accessibilityMode>
               <currencySymbol>$</currencySymbol>
               <orgAttachmentFileSizeLimit>5242880</orgAttachmentFileSizeLimit>
               <orgDefaultCurrencyIsoCode>USD</orgDefaultCurrencyIsoCode>
               <orgDisallowHtmlAttachments>false</orgDisallowHtmlAttachments>
               <orgHasPersonAccounts>false</orgHasPersonAccounts>
               <organizationId>00D36000000wCdEEEE</organizationId>
               <organizationMultiCurrency>false</organizationMultiCurrency>
               <organizationName>Pirates, Inc.</organizationName>
               <profileId>00e36000001JVJdAAO</profileId>
               <roleId xsi:nil="true"/>
               <sessionSecondsValid>7200</sessionSecondsValid>
               <userDefaultCurrencyIsoCode xsi:nil="true"/>
               <userEmail>pirate_jack@trailhead-pirates.org</userEmail>
               <userFullName>Pirate Jack</userFullName>
               <userId>00536000002BU42AAG</userId>
               <userLanguage>en_US</userLanguage>
               <userLocale>en_US</userLocale>
               <userName>pirate_jack@trailhead-pirates.org</userName>
               <userTimeZone>America/Los_Angeles</userTimeZone>
               <userType>Standard</userType>
               <userUiSkin>Theme3</userUiSkin>
            </userInfo>
         </result>
      </loginResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Congratulations, captain. You successfully logged in. The response contains a bunch of information about your org and user. Most importantly, it contains your org's instance (or custom domain, if you're using My Domain) and a session ID that we'll use to make future requests, highlighted here.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns="urn:enterprise.soap.sforce.com"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <loginResponse>
         <result>
            <metadataServerUrl>https://na30.salesforce.com/services/Soap/m/36.0/00D36000000wCdk</metadataServerUrl>
            <passwordExpired>false</passwordExpired>
            <sandbox>false</sandbox>
            <serverUrl>https://na30.salesforce.com/services/Soap/c/36.0/00D36000000wCdk</serverUrl>
            <sessionId>00D36000000wCdk!AQEAQK23h7Ak_SyGTu_WrzC1a2KTGnZacSOWiR4jc8Mp2Jt7_f4t8XkJ.g64WlP7_JfweOHn.Ak.SwUEkQR0XGcKAxKq4gtU</sessionId>
            <userId>00536000002BU42AAG</userId>
```

Copy the instance and session ID into a text file. We'll need them in a minute.

Because your org's instance is likely to change, don't hardcode references to the instance when you start building integrations! Instead, use the Salesforce feature My Domain to configure a custom domain. A custom domain not only eliminates the headache of changing instance names, you can use it to highlight your brand, make your org more secure, and personalize your login page.

Create an Account

Just like we did with REST API, let's create an account with SOAP API. In the navigation pane on the left side of the screen, find the `create` operation. Expand it, and double-click **Request 1**.

Because creating a record is more complicated than logging in, the `create()` SOAP message includes many more elements. Most of the elements are contained in the request header, and most of them are optional. To simplify things, we'll delete most of the header info, but remember that the options provided by these headers are available when you create a record. For information on what each header does, check out the SOAP Headers topic in the SOAP API Developer Guide.

One header we don't want to delete, though, is `SessionHeader`. It's going to contain the session ID we got from the `login()` response. Go ahead and delete the other headers, from to . Your message looks like this.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:urn="urn:enterprise.soap.sforce.com"
  xmlns:urn1="urn:sobject.enterprise.soap.sforce.com">
   <soapenv:Header>
      <urn:SessionHeader>
         <urn:sessionId>?</urn:sessionId>
      </urn:SessionHeader>
   </soapenv:Header>
   <soapenv:Body>
      <urn:create>
         <!--Zero or more repetitions:-->
         <urn:sObjects>
            <!--Zero or more repetitions:-->
            <urn1:fieldsToNull>?</urn1:fieldsToNull>
            <urn1:Id>?</urn1:Id>
         </urn:sObjects>
      </urn:create>
   </soapenv:Body>
</soapenv:Envelope>
```

Get the session ID you copied to a text file and paste it inside the tag, replacing the *?*.

We have a few more adjustments to make to the message body. First, we specify that we're creating an account. Change the text in the tag to look like this: . This adjustment specifies the correct record type using the XML instance schema declaration.

We also want to give the account a name. Add `Sample SOAP Account` inside the `sObjects` element. Delete the `fieldsToNull` and `Id` elements, too. Now your message looks something like this.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:urn="urn:enterprise.soap.sforce.com"
  xmlns:urn1="urn:sobject.enterprise.soap.sforce.com">
   <soapenv:Header>
      <urn:SessionHeader>
         <urn:sessionId>00D36000000wCdk!AQEAQK23h7Ak_SyGTu_WrzC1a2KTGnZacSOWiR4jc8Mp2Jt7_f4t8XkJ.g64WlP7_JfweOHn.Ak.SwUEkQR0XGcKAxKq4gtU</urn:sessionId>
      </urn:SessionHeader>
   </soapenv:Header>
   <soapenv:Body>
      <urn:create>
         <!--Zero or more repetitions:-->
         <urn:sObjects xsi:type="urn1:Account" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <!--Zero or more repetitions:-->
            <Name>Sample SOAP Account</Name>
         </urn:sObjects>
      </urn:create>
   </soapenv:Body>
</soapenv:Envelope>
```

One last thing to do before we make the request. Change the endpoint to specify your org's instance rather than `login`, and remove the package version from the end of the URI. The endpoint URI looks something like this: `https://na30.salesforce.com/services/Soap/c/36.0`.

We're ready to send the request. Click the green triangle again. It worked! Let's look at the response.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns="urn:enterprise.soap.sforce.com">
   <soapenv:Header>
      <LimitInfoHeader>
         <limitInfo>
            <current>9</current>
            <limit>15000</limit>
            <type>API REQUESTS</type>
         </limitInfo>
      </LimitInfoHeader>
   </soapenv:Header>
   <soapenv:Body>
      <createResponse>
         <result>
            <id>0013600000L0s2QAAR</id>
            <success>true</success>
         </result>
      </createResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Notice the `LimitInfoHeader`. This header returns information about your API usage. In the above example, we've made 9 out of 15,000 allowed calls today.

In the response body, notice the element. `true` indicates that the record was created successfully. includes the ID of the record, which you can use in future requests.

We covered the basics of making requests with SOAP API. Of course, each operation has its own parameters and peculiarities. Make sure that you use the SOAP API Developer Guide as your map when you start writing integrations with SOAP API.

Resources

Use Bulk API

Learning Objectives

After completing this unit, you'll be able to:

- Describe how an asynchronous request differs from a synchronous request.
- Create a bulk job using REST Explorer in Workbench.
- Import data to your Salesforce org by adding batches to a job.
- Monitor a job's progress.
- Get a batch's results.

Bulk API and Asynchronous Requests

Bulk API is based on REST principles and is optimized for working with large sets of data. You can use it to query, insert, update, upsert, or delete many records asynchronously, meaning that you submit a request and come back for the results later. Salesforce processes the request in the background.

In contrast, SOAP and REST API use synchronous requests and are optimized for real-time client applications that update a few records at a time. You can use both of these APIs for processing many records, but when the data sets contain hundreds of thousands of records, they're less practical. Bulk API's asynchronous framework is designed to make it simple and efficient to process data from a few thousand to millions of records.

The easiest way to use Bulk API is to enable it for processing records in Data Loader using CSV files. With Data Loader, you don't have to write your own client app. Sometimes, though, unique requirements necessitate writing a custom app. Bulk API lets you take the ship's wheel into your own hands and steer the course toward a solution that works for you.

Import Data to Your Org

To explore Bulk API, we'll use SoapUI and Workbench to create some account records.

Get a Session ID

Authentication works a bit differently in Bulk API. We need a valid session ID, which we'll include in the X-SFDC-Session header with each of our Bulk API requests. We obtain a session ID using the SOAP API `login()` call, just like we did in the previous unit. See Log In to Your DE Org for instructions, and make sure to copy the session ID into a text file.

Create a Bulk Job

Bulk API is REST-based, so we can use Workbench's REST Explorer to make Bulk API requests.

1. Log in to your Trailhead DE org and navigate to Workbench.
2. For Environment, select **Production**. For API Version, select the highest available number. Make sure that you select **I agree to the terms of service**.
3. Click **Login with Salesforce**.
4. In the top menu, select .

Now we're ready to upload our data. The first step is to create a job. A job specifies the type of operation and data object we're working with. It functions as a bucket into which we add batches of data for processing.

A job is represented by the JobInfo resource. This resource creates a job, gets a job's status, and changes a job's status.

To create a job, we submit a POST request to the JobInfo resource with the job's properties in the request body. Because Bulk API is REST-based, the request takes the familiar form of a REST request with four components: URI, HTTP method, headers, and body. The method is POST, as we just mentioned.

For the URI, replace the text in the URI text box with the following: `/services/async/XX.0/job`, where `XX.0` corresponds to the API version you're using. Let's note a few things about this URI.

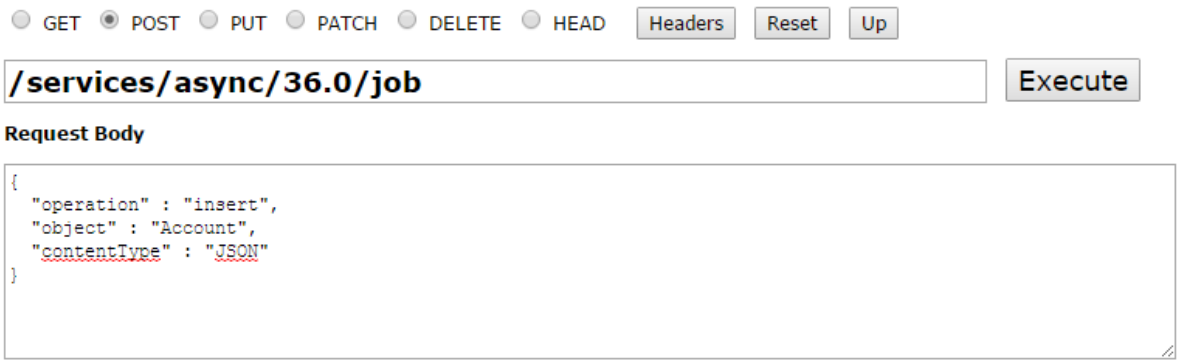
- We're using `/services/async` instead of `/services/data`. The `async` portion specifies that we're making a Bulk API request.
- Like SOAP API, the API version number doesn't include a `v` prefix.
- `/job` indicates that we're accessing the JobInfo resource.

For the request body, copy and paste the following text.

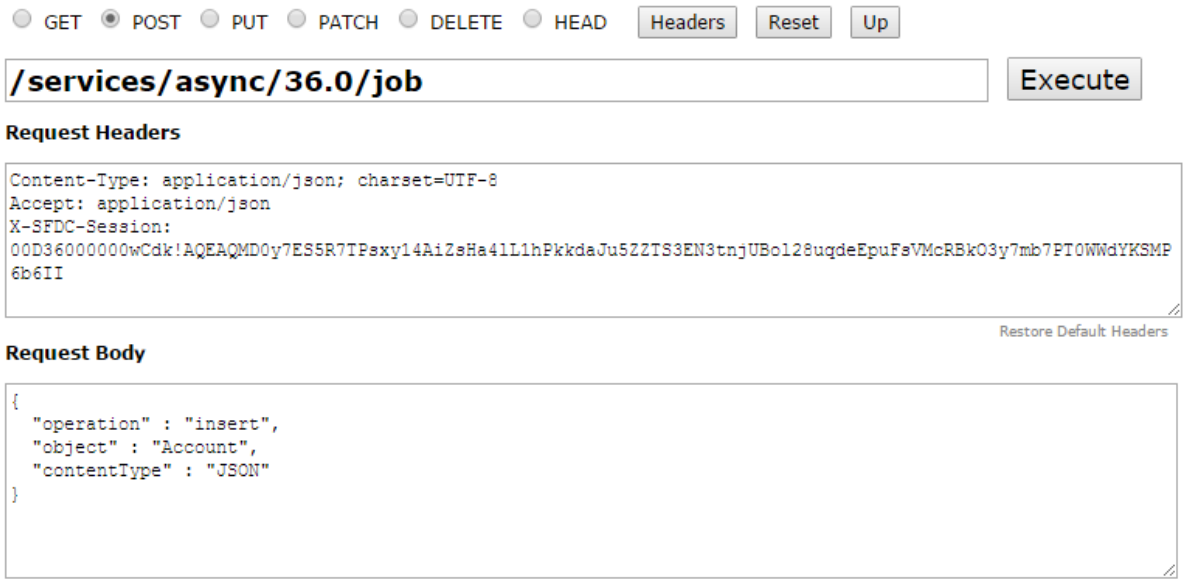
```
{
  "operation" :
"insert",
  "object" : "Account",
  "contentType" : "JSON"
}
```

These properties indicate that we want to use the insert operation on the data we submit to the job. We're submitting account data, and it's in JSON format. Bulk API supports payloads in CSV, XML, and JSON.

Your REST Explorer looks something like this.



To authenticate this request, we need to include the session ID we got from the SOAP API `login()` response. Click **Headers**. Under the `Accept` header, add the following text: `X-SFDC-Session: sessionID`. Replace *sessionID* with the session ID you copied earlier. Your REST Explorer looks something like this.



Click **Execute** and check out the response.

The response includes all sorts of properties about the job, most of which aren't useful to us right now because we haven't added batches yet. Although we want to note two properties. Look at the job ID (id) and copy it into a text file. We'll be using it to add batches to the job and check in on the job's status. The other property is state. When you create a job, it's immediately set to the `Open` state. That means it's ready to start receiving batches.

Add Batches to the Job

Now we can insert account data via a batch. A batch is a set of records sent to the server in a POST request. The server processes each batch independently, not necessarily in the order it's received. When created, the status of a batch is accessible via the BatchInfo resource.

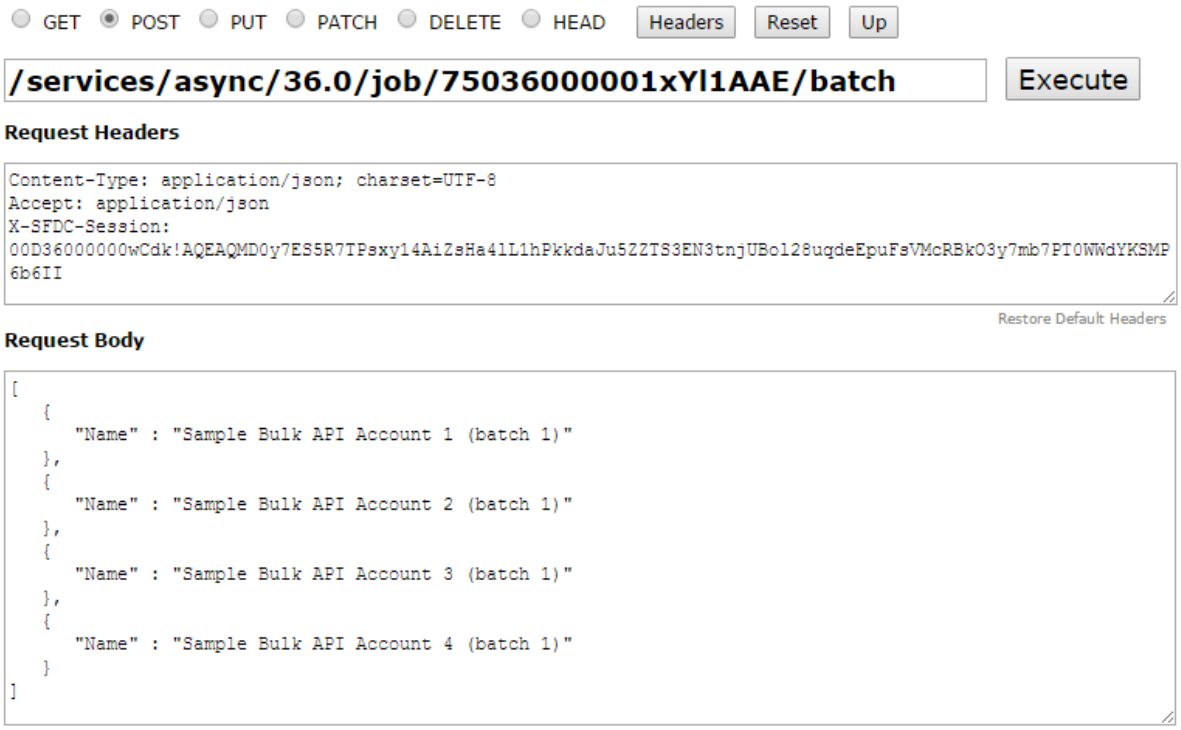
When a batch is complete, the result for each record is available in a result set resource. We'll get to that in a bit. For now, replace the text in the URI text box with the following: `/services/async/`*`XX`*`.0/job/`*`jobID`*`/batch`. Replace *jobID* with the job ID you copied.

For this example, we're adding a batch with only four accounts. Usually, you use Bulk API to add thousands or millions of records, but the principle is the same. Copy the following JSON text into the request body.

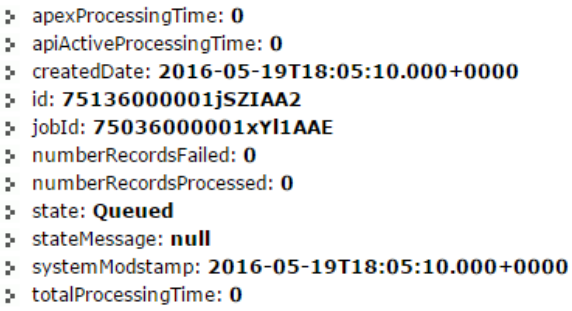
```
[
   {
      "Name" : "Sample Bulk API Account 1 (batch
1)"
   },
   {
      "Name" : "Sample Bulk API Account 2 (batch
1)"
   },
   {
      "Name" : "Sample Bulk API Account 3 (batch
1)"
   },
   {
      "Name" : "Sample Bulk API Account 4 (batch
1)"
   }
]
```

We're using JSON because that's what we specified when we created the job.

Click **Headers** and make sure that the `X-SFDC-Session` header is still there with your session ID. Your request looks something like this.



Click **Execute**. Let's examine the result.



The response contains information about the batch, including a batch ID and a state. The state is Queued. The server is aware of the request and has queued it for processing.

Let's add another batch to the same job. Replace the JSON text in the request body with the following.

```
[
   {
      "Name" : "Sample Bulk API Account 5 (batch
2)"
   },
   {
      "Name" : "Sample Bulk API Account 6 (batch
2)"
   },
   {
      "Name" : "Sample Bulk API Account 7 (batch
2)"
   },
   {
      "Name" : "Sample Bulk API Account 8 (batch
2)"
   }
]
```

Click **Execute**. We get a similar response with a different batch ID.

Close the Job

Now that we've submitted our batches, we can move on to monitoring them and retrieving our results. Before we do, though, it's a good practice to change the job's status to Closed because we aren't submitting any more batches.
Replace the text in the URI text box with the following: `/services/async/XX.0/job/jobID`. Again, replace *XX* with the API version you're using, and replace *jobID* with the job's ID. For the HTTP method, select POST. In the request body, replace the text with the following JSON text.

```
{
   "state" :
"Closed"
}
```

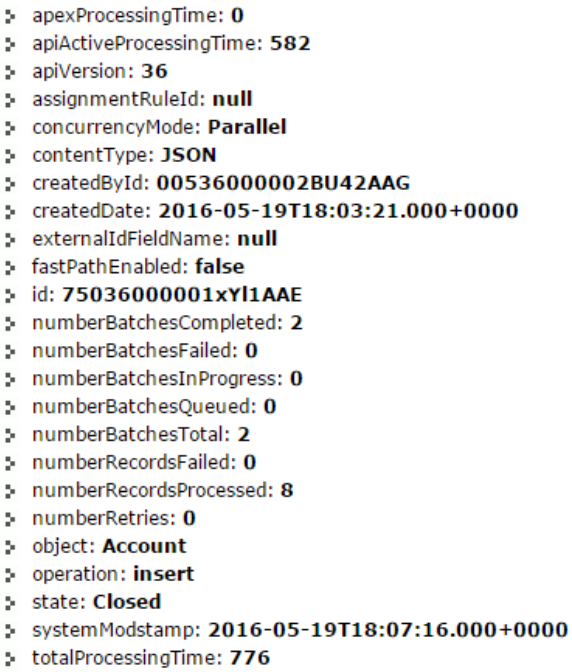
Click **Execute**. The response contains job status information. The state property indicates that the job is closed.

Check the Status of the Job

We've submitted our batches. We've closed the job. Now it's up to the server to process the request. We can monitor the server's progress by checking the status of the job's batches through the API or through the Salesforce UI. In Salesforce, from Setup, enter `Bulk Data Load Jobs` in the Quick Find box, then select **Bulk Data Load Jobs**. You can check the status of a job on this page. Or, you can click a job ID to check the statuses and get results for batches associated with that job.

In the API, we use the JobInfo resource to monitor a job. Replace the text in the URI text box with the following: `/services/async/XX.0/job/jobID`, with the usual replacements. For the HTTP method, select GET.

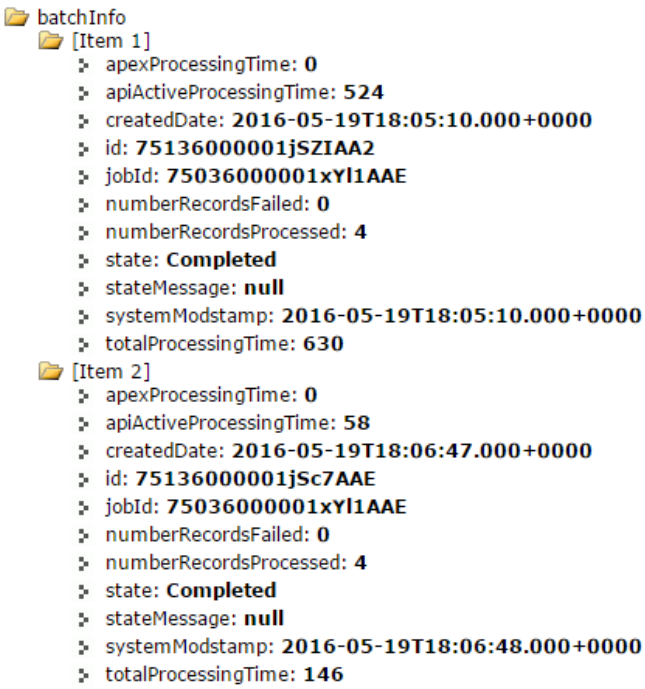
Click **Execute**. You see something like this.



If your numberBatchesQueued is 1 or 2, at least one of your batches is still in the queue. Don't worry, it'll be processed in a few minutes. In the meantime, go treat yourself to a mug of grog and try the same request again when you get back. If you're lucky and your batches are already processed, continue on to retrieve the batch results, or feel free to take a grog break anyway.

Get the Batch Results

First, let's check the statuses of the individual batches to confirm what we learned from the job's status. Add `/batch` to the end of the URI in the URI text box—this is the URI for the BatchInfo resource. Click **Execute**. Expand the **batchInfo** folder, and then expand the **[Item 1]** and **[Item 2]** folders. Your results look something like this.



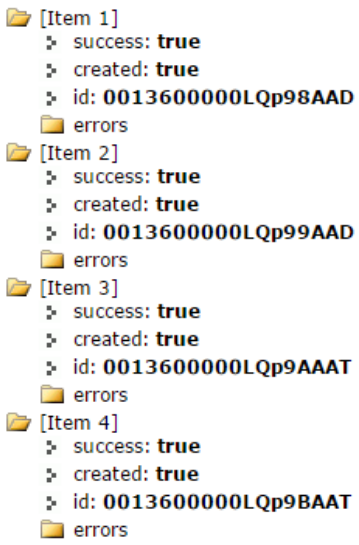
As we determined from the JobInfo resource, each batch has completed. Next, we access the results resource to get the results of the first batch.

Copy the value next to the id field under **Item 1**. Make sure that you copy the id and not the jobId. In the URI text box, after `/batch`, enter `/` and paste the ID value. Then add `/result`.

The URI looks something like this: `/services/async/36.0/job/75036000001xYl1AAE/batch/75136000001jSZIAA2/result`.

Click **Execute**. Expand each **[Item]** folder in the response. The results look something like this.

Each result contains all the information we get when we create a record through the API, including the new record's ID. To get the other batch's results, in the URI, replace this batch's ID with the other batch's ID. Tidy work, captain!



Resources

Use Streaming API

Learning Objectives

After completing this unit, you'll be able to:

- Describe the primary benefit that push technology offers over pull technology.
- Create a PushTopic and receive event notifications.
- Specify replay options for durable streaming.
- Broadcast a message with generic streaming.

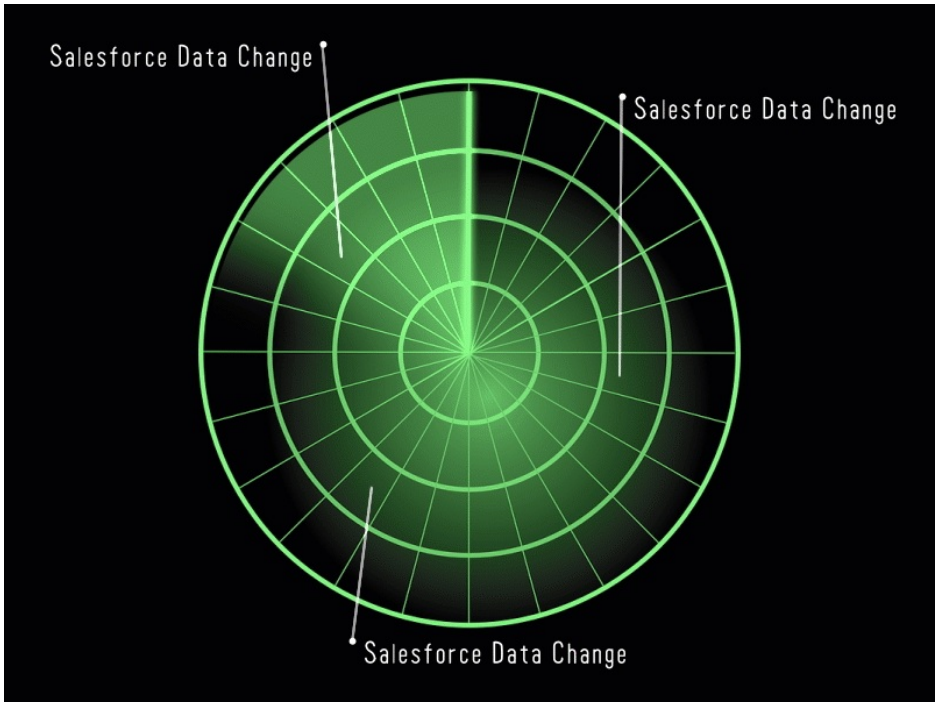
Streaming Events

To conclude our survey of Salesforce's data APIs, let's look at an API that serves an entirely different use case. Streaming API lets you push a stream of notifications from Salesforce to client apps based on criteria that you define. How does pushing notifications differ from the pull paradigm that our other APIs use, in which the client app requests, or pulls, data from Salesforce? Let's examine the problem from a ship captain's point of view.

Imagine that you're sailing the high seas, and you want to keep an eye out for oncoming hazards, other ships, and islands rich with treasure. You put a sailor in the crow's nest to keep an active lookout. Now put on your developer hat again. Let's say you're writing an app using REST or SOAP API that periodically checks to see if any accounts have been updated. You can use a similar solution and keep an active lookout by constantly requesting account data and checking to see if it matches the old data.

Now imagine that you're on your ship again, but this time you have access to a shiny, new radar display. You don't need to keep a sailor in the crow's nest, because whenever an object of interest approaches, the display beeps.

Streaming API is your radar. It lets you define events and push notifications to your client app when the events occur. You don't have to keep an active lookout for data changes—you don't have to constantly poll Salesforce and make unnecessary API requests.



Tracking data changes in Salesforce is especially useful when you have business data stored in a system external to Salesforce. You can use Streaming API to keep your external source in sync with your Salesforce data. Also, Streaming API lets you process business logic in an external system in response to data changes in Salesforce. For example, you can use Streaming API to notify a fulfillment center whenever an opportunity is updated.

In addition to data changes, you can use Streaming API to broadcast notifications for events defined outside of Salesforce. For example, you can make your app display a message whenever a system maintenance window is about to start or when a new offer is available to your users. This functionality is called generic streaming, and we'll cover it as well.

Streaming API uses the Bayeux protocol and the CometD messaging library.

PushTopics

We interface with Streaming API through PushTopics. A PushTopic is an sObject that contains the criteria of events you want to listen to, such as data changes for a particular object. You define the criteria as a SOQL query in the PushTopic and specify the record operations to notify on (create, update, delete, and undelete). In addition to event criteria, a PushTopic represents the channel that client apps subscribe to.

We'll dive deeper when we create our own PushTopic.

Supported Objects in PushTopic Queries

PushTopic queries support all custom objects. PushTopic queries support the following standard objects.

- Account
- Campaign
- Case
- Contact
- Lead
- Opportunity
- Task

The following standard objects are supported in PushTopic queries through a pilot program.

- ContractLineItem
- Entitlement
- LiveChatTranscript
- Quote
- QuoteLineItem
- ServiceContract

PushTopics and Notifications

A PushTopic enables you to define the object, fields, and criteria you're interested in receiving event notifications for. The following example shows a PushTopic defined and inserted in Apex. After this PushTopic is created, you can subscribe to this PushTopic channel to track changes on accounts whose billing city is San Francisco. This PushTopic specifies that the Id, Name, Phone fields are returned in each event notification. By default, notifications are sent for create, update, delete, and undelete operations that match the query's criteria.

```
PushTopic pushTopic = new PushTopic();
pushTopic.Name = 'AccountUpdates';
pushTopic.Query = 'SELECT Id, Name, Phone FROM Account WHERE BillingCity=\'San
Francisco\'';
pushTopic.ApiVersion = 37.0;

insert pushTopic;
```

At the minimum, define the PushTopic name, query, and API version. You can use default values for the remaining properties. By default, the fields in the `SELECT` statement field list and `WHERE` clause are the ones that trigger notifications. Notifications are sent only for the records that match the criteria in the `WHERE` clause. To change which fields trigger notifications, set `pushTopic.NotifyForFields` to one of these values.

NotifyForFields Value	Description
`All`	Notifications are generated for all record field changes, provided the evaluated records match the criteria specified in the WHERE clause.
`Referenced (default)`	Changes to fields referenced in the SELECT and WHERE clauses are evaluated. Notifications are generated for the evaluated records only if they match the criteria specified in the WHERE clause.
`Select`	Changes to fields referenced in the SELECT clause are evaluated. Notifications are generated for the evaluated records only if they match the criteria specified in the WHERE clause.
`Where`	Changes to fields referenced in the WHERE clause are evaluated. Notifications are generated for the evaluated records only if they match the criteria specified in the WHERE clause.

To set notification preferences explicitly, set the following properties to either `true` or `false`. By default, all values are set to `true`.

```
pushTopic.NotifyForOperationCreate = true;
pushTopic.NotifyForOperationUpdate = true;
pushTopic.NotifyForOperationUndelete =
true;
pushTopic.NotifyForOperationDelete = true;
```

If you create an account, an event notification is generated. The notification is in JSON and contains the fields that we specified in the PushTopic query: Id, Name, and Phone. The event notification looks similar to the following.

```
{
  "clientId": "lxdl9o32njygi1gj47kgfaga4k",
  "data": {
    "event": {
      "createdDate": "2016-09-16T19:45:27.454Z",
      "replayId": 1,
      "type": "created"
    },
    "sobject": {
      "Phone": "(415) 555-1212",
      "Id": "001D000000KneakIAB",
      "Name": "Blackbeard"
    }
  },
  "channel": "/topic/AccountUpdates"
}
```

The notification message includes the channel for the PushTopic, whose name format is `/topic/`*`PushTopicName`*. When you create a PushTopic, the channel is created automatically.

PushTopic Queries

Let's take a moment to dive into the query we just defined for our PushTopic. PushTopic queries are regular SOQL queries, so if you're familiar with SOQL, you don't need to learn a new format. If you're not familiar, don't worry. We cover the basic query format here. In short, the query contains a `SELECT` statement with an optional `WHERE` clause, as follows.

```
SELECT <comma-separated list of fields> FROM <Salesforce object> WHERE <filter
criteria>
```

By default, the comma-separated list of fields specifies the fields to detect changes for. The values of these fields are sent in the notification messages. The `FROM` clause specifies the object whose data changes we're interested in. The `WHERE` clause narrows down the set of records to a more specific set based on criteria. A change to a field in the `WHERE` clause also generates a notification if the conditions in the `WHERE` clause are met. You can change which fields generate notifications by modifying the PushTopic field NotifyForFields.
The `WHERE` clause supports operators that SOQL supports, such as the `LIKE`, `IN`, `OR`, and `AND` operators. For example, to filter on accounts whose billing city is either San Francisco or New York, we'd use this query.

```
SELECT Id, Name, Phone FROM Account WHERE BillingCity='San Francisco' OR BillingCity='New
York'
```

To ensure that notifications are sent in a timely manner, the following requirements apply to PushTopic queries.

- The `SELECT` statement's field list must include Id.
- Only one object per query is allowed.
- The object must be valid for the specified API version.

Certain queries aren't supported, such as aggregate queries or semi-joins.

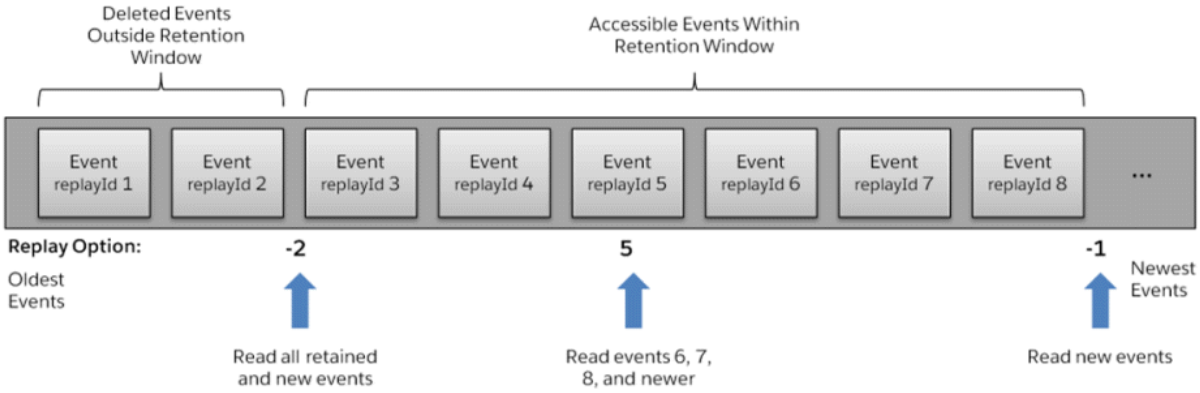
Retrieve Past Notifications Using Durable Streaming

So far, you've learned about PushTopics and event notifications for Salesforce record changes. What happens if a Salesforce record is created or updated before a client subscribes to the PushTopic? Before API version 37.0, the client misses the corresponding notification. As of API version 37.0, Salesforce stores events that match PushTopic queries, even if no one is subscribed to the PushTopic. The events are stored for 24 hours, and you can retrieve them at any time during that window. Yay!

Starting with API version 37.0, each event notification contains a field called replayId. Similar to replaying a video, Streaming API replays the event notifications that were sent by using the replayId field. The value of the replayId field is a number that identifies the event in the stream. The replay ID is unique for the org and channel. When you replay an event, you're retrieving a stored event from a location in the stored stream. You can either retrieve a stream of events starting after the event specified by a replay ID, or you can retrieve all stored events. Here's a summary of the replay options we can specify when subscribing to a channel.

Replay Option	Description
replayId	Subscriber receives notifications for events that occur after the event specified by the replay ID value.
-1	Subscriber receives notifications for all new events that occur after subscription.
-2	Subscriber receives notifications for all new events that occur after subscription and previous events that fall within the 24-hour retention window.

This diagram shows how event consumers can read a stream of events by using various replay options.



Sample Durable Streaming Apps

To learn how to subscribe to events and use replay options, check out the durable streaming code examples in the Streaming API Developer Guide. Several examples are provided including a Visualforce page that uses a JavaScript CometD extension, and a Java client that uses a CometD connector. Links to these code examples are included in the Resources section.

Generic Streaming

Before you set sail on your own, let's spend a few minutes reviewing generic streaming. Streaming API supports sending notifications with a generic payload that aren't tied to Salesforce data changes.
You can use generic streaming for any situation where you want to send custom notifications, such as:

- Broadcasting messages to specific teams or to your entire organization
- Sending notifications for events that are external to Salesforce

To use generic streaming, you need:

- A streaming channel that defines the channel
- One or more clients subscribed to the channel
- The Streaming Channel Push resource to monitor and invoke events on the channel

You can create a streaming channel for generic streaming either through the Streaming Channels app in the user interface, or through the API. A streaming channel is represented by the StreamingChannel sObject, so you can create it through Apex, REST API, or SOAP API. The format of the channel name for generic streaming is `/u/ChannelName`. For example, this Apex snippet creates a channel named `Broadcast`.

```
StreamingChannel ch = new
StreamingChannel();
ch.Name = '/u/Broadcast';

insert ch;
```

Alternatively, you can opt to have Salesforce create the streaming channel dynamically for you if it doesn't exist. To enable dynamic streaming channels in your org, from Setup, enter `User Interface` in the Quick Find box, then select **User Interface**. On the User Interface page, select the **Enable Dynamic Streaming Channel Creation** option.

You can subscribe to the channel by using a CometD client. (The Resources section links to a sample walkthrough in the Streaming API Developer Guide.)

To generate events, make a POST request to the following REST resource. Replace *XX.0* with the API version and *Streaming Channel ID* with the ID of your channel.

```
/services/data/vXX.0/sobjects/StreamingChannel/Streaming Channel
ID/push
```

Note

To obtain your channel ID, run a SOQL query on StreamingChannel, such as: `SELECT Id, Name FROM StreamingChannel`

Example REST request body.

```
{
  "pushEvents": [
      {
          "payload": "Broadcast message to all
subscribers",
          "userIds": []
      }
   ]
}
```

Note

Instead of broadcasting to all subscribers, specify a list of subscribed users to send notifications to by using the optional `userIds` field. Also, you can use the GET method of the Streaming Channel Push REST API resource to get a list of active subscribers to the channel.

The event notification that the subscribed client receives looks similar to the following.

```
{
  "clientId": "1p145y6g3x3nmnlodd7v9nhi4k",
  "data": {
    "payload": "Broadcast message to all
subscribers",
    "event": {
      "createdDate": "2016-09-16T20:43:39.392Z",
      "replayId": 1
    }
  },
  "channel": "/u/Broadcast"
}
```

Notice that this event notification contains the replayId field. As with PushTopic streaming, generic event notifications are also stored for 24 hours and can be retrieved using the replayId value starting in API version 37.0.

Thanks to event notifications, you can set sail in the high seas with confidence and head to the island rich with treasure!

Apex Integration Overview

Learning Objectives

After completing this module, you will be able to:

- Describe the differences between web service and HTTP callouts.
- Authorize an external site with remote site settings.

Make Callouts to External Services from Apex

An Apex callout enables you to tightly integrate your Apex code with an external service. The callout makes a call to an external web service or sends an HTTP request from Apex code, and then receives the response.

Apex callouts come in two flavors.

- Web service callouts to SOAP web services use XML, and typically require a WSDL document for code generation.
- HTTP callouts to services typically use REST with JSON.

These two types of callouts are similar in terms of sending a request to a service and receiving a response. But while WSDL-based callouts apply to SOAP Web services, HTTP callouts can be used with any HTTP service, either SOAP or REST.

So you are probably asking yourself right now, “Which one should I use?” Whenever possible, use an HTTP service. These services are typically easier to interact with, require much less code, and utilize easily readable JSON. All the “cool kids” have been switching to REST services over the last couple of years, but that’s not to say that SOAP Web services are bad. They’ve been around forever (in Internet years) and are commonly used for enterprise applications. They are not going away anytime soon. You’ll probably use SOAP mostly when integrating with legacy applications or for transactions that require a formal exchange format or stateful operations. In this module we’ll touch on SOAP, but will spend most of our time on REST.

Authorize Endpoint Addresses

We love security at Salesforce! So, any time you make a callout to an external site we want to make sure that it is authorized. We can’t have code calling out willy-nilly to any endpoint without prior approval. Before you start working with callouts, update the list of approved sites for your org on the Remote Site Settings page.

We’ll be using the following endpoints in this module, so go ahead and add them now. If you forget to add an endpoint, trust me, you'll get a reminder when you try to run your code. We’ll be making calls to the following sites.

- https://th-apex-http-callout.herokuapp.com
- https://th-apex-soap-service.herokuapp.com

Authorize both of these endpoint URLs by following these steps.

1. From Setup, enter `Remote Site Settings` in the Quick Find box, then click **Remote Site Settings**.
2. Click **New Remote Site**.
3. For the remote site name, enter `animals_http`.

4. For the remote site URL, enter `https://th-apex-http-callout.herokuapp.com`. This URL authorizes all subfolders for the endpoint, like https://th-apex-http-callout.herokuapp.com/path1 and https://th-apex-http-callout.herokuapp.com/path2.
5. For the description, enter `Trailhead animal service: HTTP`.
6. Click **Save & New**.
7. For the second remote site name, enter `animals_soap`.
8. For the remote site URL, enter `https://th-apex-soap-service.herokuapp.com`.
9. For the description, enter `Trailhead animal service: SOAP`.
10. Click **Save**.

Apex REST Callouts

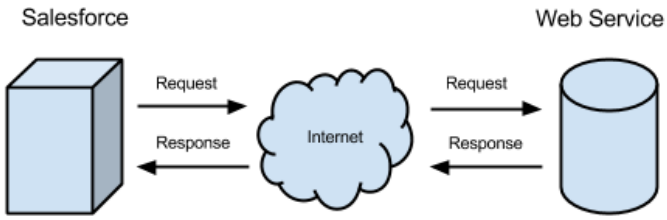
Learning Objectives

After completing this module, you'll be able to:

- Perform a callout to receive data from an external service.
- Perform a callout to send data to an external service.
- Test callouts by using mock callouts.

HTTP and Callout Basics

REST callouts are based on HTTP. To understand how callouts work, it's helpful to understand a few things about HTTP. Each callout request is associated with an HTTP method and an endpoint. The HTTP method indicates what type of action is desired.



The simplest request is a GET request (GET is an HTTP method). A GET request means that the sender wants to obtain information about a resource from the server. When the server receives and processes this request, it returns the request information to the recipient. A GET request is similar to navigating to an address in the browser. When you visit a web page, the browser performs a GET request behind the scenes. In the browser, the result of the navigation is a new HTML page that's displayed. With a callout, the result is the response object.

To illustrate how a GET request works, open your browser and navigate to the following URI: https://th-apex-http-callout.herokuapp.com/animals. Your browser displays a list of animals in a weird format, because the service returns the response in a format called JSON. Sometimes a GET response is also formatted in XML.

The following are descriptions of common HTTP methods.

Table 1. Some Common HTTP Methods

HTTP Method	Description
GET	Retrieve data identified by a URL.
POST	Create a resource or post data to the server.
DELETE	Delete a resource identified by a URL.
PUT	Create or replace the resource sent in the request body.

If you have some free time, browse the exhaustive list of all HTTP methods in the Resources section.

In addition to the HTTP method, each request sets a URI, which is the endpoint address at which the service is located. For example, an endpoint can be http://www.example.com/api/resource. In the example in the "HTTP and Callout Basics" unit, the endpoint is https://th-apex-http-callout.herokuapp.com/animals.

When the server processes the request, it sends a status code in the response. The status code indicates whether the request was processed

successfully or whether errors were encountered. If the request is successful, the server sends a status code of `200`. You've probably seen some other status codes, such as 404 for file not found or 500 for an internal server error.

If you still have free time after browsing the list of HTTP methods, check out the list of all response status codes in the Resources section. If you're having a hard time sleeping at night, these two resources can help.

Note

In addition to the endpoint and the HTTP method, you can set other properties for a request. For example, a request can contain headers that provide more information about the request, such as the content type. A request can also contain data to be sent to the service, such as for POST requests. You'll see an example of a POST request in a later step. A POST request is similar to clicking a button on a web page to submit form data. With a callout, you send data as part of the body of the request instead of manually entering the data on a web page.

Get Data from a Service

It's time to put your new HTTP knowledge to use with some Apex callouts. This example sends a GET request to a web service to get a list of woodland creatures. The service sends the response in JSON format. JSON is essentially a string, so the built-in `JSONParser` class converts it to an object. We can then use that object to write the name of each animal to the debug log.

Before you run the examples in this unit, you'll need to authorize the endpoint URL of the callout, https://th-apex-http-callout.herokuapp.com, using the steps from the Authorize Endpoint Addresses section.

1. Open the Developer Console under Your Name or the quick access menu ( ).
2. In the Developer Console, select .
3. Delete the existing code and insert the following snippet.

```
Http http = new Http();
HttpRequest request = new HttpRequest();
request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
request.setMethod('GET');
HttpResponse response = http.send(request);
// If the request is successful, parse the JSON response.
if (response.getStatusCode() == 200) {
    // Deserialize the JSON string into collections of primitive data types.
    Map<String, Object> results = (Map<String, Object>)
JSON.deserializeUntyped(response.getBody());
    // Cast the values in the 'animals' key as a list
    List<Object> animals = (List<Object>) results.get('animals');
    System.debug('Received the following animals:');
    for (Object animal: animals) {
        System.debug(animal);
    }
}
```

4. Select **Open Log**, and then click **Execute**.
5. After the debug log opens, select **Debug Only** to view the output of the System.debug statements.

   The names of the animals are displayed.

The JSON for our example is fairly simple and easy to parse. For more complex JSON structures, you can use JSON2Apex. This tool generates strongly typed Apex code for parsing a JSON structure. You simply paste in the JSON, and the tool generates the necessary Apex code for you. Brilliant!

Send Data to a Service

Another common use case for HTTP callouts is sending data to a service. For instance, when you buy the latest Justin Bieber album or comment on your favorite "Cat in a Shark Costume Chases a Duck While Riding a Roomba" video, your browser is making a POST request to submit data. Let's see how we send data in Apex.

This example sends a POST request to the web service to add an animal name. The new name is added to the request body as a JSON string. The request `Content-Type` header is set to let the service know that the sent data is in JSON format so that it can process the data appropriately. The service responds by sending a status code and a list of all animals, including the one you added. If the request was processed successfully, the status code returns 201, because a resource has been created. The response is sent to the debug log when anything other than 201 is returned.

1. Open the Developer Console under Your Name or the quick access menu ( ).

2. In the Developer Console, select .
3. Delete any existing code and insert the following snippet.

```
Http http = new Http();
HttpRequest request = new HttpRequest();
request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
request.setMethod('POST');
request.setHeader('Content-Type', 'application/json;charset=UTF-8');
// Set the body as a JSON object
request.setBody('{"name":"mighty moose"}');
HttpResponse response = http.send(request);
// Parse the JSON response
if (response.getStatusCode() != 201) {
    System.debug('The status code returned was not expected: ' +
        response.getStatusCode() + ' ' + response.getStatus());
} else {
    System.debug(response.getBody());
}
```

4. Select **Open Log**, and then click **Execute**.
5. When the debug log opens, select **Debug Only** to view the output of the `System.debug` statement. The last item in the list of animals is `"mighty moose"`.

Test Callouts

There's good news and bad news regarding callout testing. The bad news is that Apex test methods don't support callouts, and tests that perform callouts fail. The good news is that the testing runtime allows you to "mock" the callout. Mock callouts allow you to specify the response to return in the test instead of actually calling the web service. You are essentially telling the runtime, "I know what this web service will return, so instead of calling it during testing, just return this data." Using mock callouts in your tests helps ensure that you attain adequate code coverage and that no lines of code are skipped due to callouts.

Prerequisites

Before you write your tests, let's create a class that contains the GET and POST request examples we executed anonymously in the "Send Data to a Service" unit. These examples are slightly modified so that the requests are in methods and return values, but they're essentially the same as the previous examples.

1. In the Developer Console, select .
2. For the class name, enter `AnimalsCallouts` and then click **OK**.
3. Replace the autogenerated code with the following class definition.

```
public class AnimalsCallouts {

    public static HttpResponse makeGetCallout() {
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
        request.setMethod('GET');
        HttpResponse response = http.send(request);
        // If the request is successful, parse the JSON response.
        if (response.getStatusCode() == 200) {
            // Deserializes the JSON string into collections of primitive data types.
            Map<String, Object> results = (Map<String, Object>)
JSON.deserializeUntyped(response.getBody());
            // Cast the values in the 'animals' key as a list
            List<Object> animals = (List<Object>) results.get('animals');
            System.debug('Received the following animals:');
            for (Object animal: animals) {
                System.debug(animal);
            }
        }
        return response;
    }

    public static HttpResponse makePostCallout() {
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
        request.setMethod('POST');
        request.setHeader('Content-Type', 'application/json;charset=UTF-8');
        request.setBody('{"name":"mighty moose"}');
        HttpResponse response = http.send(request);
        // Parse the JSON response
        if (response.getStatusCode() != 201) {
            System.debug('The status code returned was not expected: ' +
                response.getStatusCode() + ' ' + response.getStatus());
        } else {
            System.debug(response.getBody());
        }
        return response;
    }

}
```

4. Press **CTRL+S** to save.

Test a Callout with StaticResourceCalloutMock

To test your callouts, use mock callouts by either implementing an interface or using static resources. In this example, we use static resources and a mock interface later on. The static resource contains the response body to return. Again, when using a mock callout, the request isn't sent to the endpoint. Instead, the Apex runtime knows to look up the response specified in the static resource and return it instead. The `Test.setMock` method informs the runtime that mock callouts are used in the test method. Let's see mock callouts in action. First, we create a static resource containing a JSON-formatted string to use for the GET request.

1. In the Developer Console, select .
2. For the name, enter `GetAnimalResource`.
3. For the MIME type, select **text/plain**, even though we are using JSON.
4. Click **Submit**.
5. In the tab that opens for the resource, insert the following content. Make sure that it is all on one line and doesn't break to the next line. This content is what the mock callout returns. It's an array of three woodland creatures.

```
{"animals": ["pesky porcupine", "hungry hippo", "squeaky
squirrel"]}
```

6. Press `CTRL+S` to save.

You've successfully created your static resource! Now, let's add a test for our callout that uses this resource.

1. In the Developer Console, select .
2. For the class name, enter `AnimalsCalloutsTest` and then click **OK**.

3. Replace the autogenerated code with the following test class definition.

```
@isTest
private class AnimalsCalloutsTest {

    @isTest static  void testGetCallout() {
        // Create the mock response based on a static resource
        StaticResourceCalloutMock mock = new
StaticResourceCalloutMock();
        mock.setStaticResource('GetAnimalResource');
        mock.setStatusCode(200);
        mock.setHeader('Content-Type', 'application/json;charset=UTF-
8');
        // Associate the callout with a mock response
        Test.setMock(HttpCalloutMock.class, mock);
        // Call method to test
        HttpResponse result = AnimalsCallouts.makeGetCallout();
        // Verify mock response is not null
        System.assertNotEquals(null,result,
            'The callout returned a null response.');
        // Verify status code
        System.assertEquals(200,result.getStatusCode(),
          'The status code is not 200.');
        // Verify content type
        System.assertEquals('application/json;charset=UTF-8',
          result.getHeader('Content-Type'),
          'The content type value is not expected.');
        // Verify the array contains 3 items
        Map<String, Object> results = (Map<String, Object>)
            JSON.deserializeUntyped(result.getBody());
        List<Object> animals = (List<Object>) results.get('animals');
        System.assertEquals(3, animals.size(),
          'The array should only contain 3 items.');
    }

}
```

4. Press **CTRL+S** to save.
5. Select . If you don't select Always Run Asynchronously, test runs that include only one class run synchronously. You can open logs from the Tests tab only for synchronous test runs.
6. To run the test, select .
7. From the Test Classes list, select **AnimalsCalloutsTest**.
8. Click .

The test result is displayed in the Tests tab under a test run ID. When the test execution finishes, expand the test run to view details. Now double-click **`AnimalCallouts`** in the Overall Code Coverage pane to see which lines are covered by your tests.

Test a Callout with HttpCalloutMock

To test your POST callout, we provide an implementation of the `HttpCalloutMock` interface. This interface enables you to specify the response that's sent in the `respond` method. Your test class instructs the Apex runtime to send this fake response by calling `Test.setMock` again. For the first argument, pass `HttpCalloutMock.class`. For the second argument, pass a new instance of `AnimalsHttpCalloutMock`, which is your interface implementation of `HttpCalloutMock`. (We'll write `AnimalsHttpCalloutMock` in the example after this one.)

```
Test.setMock(HttpCalloutMock.class, new
AnimalsHttpCalloutMock());
```

Now add the class that implements the `HttpCalloutMock` interface to intercept the callout. If an HTTP callout is invoked in test context, the callout is not made. Instead, you receive the mock response that you specify in the `respond` method implementation in `AnimalsHttpCalloutMock`.

1. In the Developer Console, select .
2. For the class name, enter `AnimalsHttpCalloutMock` and then click **OK**.
3. Replace the autogenerated code with the following class definition.

```
@isTest
global class AnimalsHttpCalloutMock implements HttpCalloutMock {
    // Implement this interface method
    global HTTPResponse respond(HTTPRequest request) {
        // Create a fake response
        HttpResponse response = new HttpResponse();
        response.setHeader('Content-Type', 'application/json');
        response.setBody('{"animals": ["majestic badger", "fluffy bunny", "scary bear", "chicken", "mighty
moose"]}');
        response.setStatusCode(200);
        return response;
    }
}
```

4. Press **CTRL+S** to save.

In your test class, create the `testPostCallout` method to set the mock callout, as shown in the next example. The `testPostCallout` method calls `Test.setMock`, and then calls the `makePostCallout` method in the `AnimalsCallouts` class. It then verifies that the response that's returned is what you specified in the respond method of the mock implementation.

1. Modify the test class `AnimalsCalloutsTest` to add a second test method. Click the class tab, and then add the following method before the closing bracket.

```
@isTest static void testPostCallout() {
    // Set mock callout class
    Test.setMock(HttpCalloutMock.class, new AnimalsHttpCalloutMock());
    // This causes a fake response to be sent
    // from the class that implements HttpCalloutMock.
    HttpResponse response = AnimalsCallouts.makePostCallout();
    // Verify that the response received contains fake values
    String contentType = response.getHeader('Content-Type');
    System.assert(contentType == 'application/json');
    String actualValue = response.getBody();
    System.debug(response.getBody());
    String expectedValue = '{"animals": ["majestic badger", "fluffy bunny", "scary bear", "chicken",
"mighty moose"]}';
    System.assertEquals(actualValue, expectedValue);
    System.assertEquals(200, response.getStatusCode());
}
```

2. Press **CTRL+S** to save.
3. Select .
4. From the Test Classes list, select **AnimalsCalloutsTest**.
5. Click .

   The test result displays in the Tests tab under a new test run ID. When the test execution finishes, expand the test run to view details about both tests.

Tell Me More...

Learn about using callouts in triggers and in asynchronous Apex, and about making asynchronous callouts.

When making a callout from a method, the method waits for the external service to send back the callout response before executing subsequent lines of code. Alternatively, you can place the callout code in an asynchronous method that's annotated with `@future(callout=true)` or use Queueable Apex. This way, the callout runs on a separate thread, and the execution of the calling method isn't blocked.

When making a callout from a trigger, the callout must not block the trigger process while waiting for the response. For the trigger to be able to make a callout, the method containing the callout code must be annotated with `@future(callout=true)` to run in a separate thread.

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Apex SOAP Callouts

Learning Objectives

After completing this module, you'll be able to:

- Generate Apex classes using WSDL2Apex.
- Perform a callout to send data to an external service using SOAP.
- Test callouts by using mock callouts.

Use WSDL2Apex to Generate Apex Code

In addition to REST callouts, Apex can also make callouts to SOAP web services using XML. Working with SOAP can be a painful (but necessary) experience. Fortunately, we have tools to make the process easier.

WSDL2Apex automatically generates Apex classes from a WSDL document. You download the web service's WSDL file, and then you upload the WSDL and WSDL2Apex generates the Apex classes for you. The Apex classes construct the SOAP XML, transmit the data, and parse the response XML into Apex objects. Instead of developing the logic to construct and parse the XML of the web service messages, let the Apex classes generated by WSDL2Apex internally handle all that overhead. If you are familiar with WSDL2Java or with importing a WSDL as a Web Reference in .NET, this functionality is similar to WSDL2Apex. You're welcome.

Note

Use outbound messaging to handle integration solutions when possible. Use callouts to third-party web services only when necessary.

For this example, we're using a simple calculator web service to add two numbers. It's a groundbreaking service that is all the rage! The first thing we need to do is download the WSDL file to generate the Apex classes. Click this link and download the calculator.xml file to your computer. Remember where you save this file, because you need it in the next step.

Generate an Apex Class from the WSDL

1. From Setup, enter `Apex Classes` in the Quick Find box, then click **Apex Classes**.
2. Click **Generate from WSDL**.
3. Click **Choose File** and select the downloaded calculator.xml file.
4. Click **Parse WSDL**. The application generates a default class name for each namespace in the WSDL document and reports any errors.

   For this example, use the default class name. However, in real life it is highly recommended that you change the default names to make them easier to work with and make your code more intuitive.

   It's time to talk honestly about the WSDL parser. WSDL2Apex parsing is a notoriously fickle beast. The parsing process can fail for several reasons, such as an unsupported type, multiple bindings, or unknown elements. Unfortunately, you could be forced to manually code the Apex classes that call the web service or use HTTP.
5. Click **Generate Apex code**. The final page of the wizard shows the generated classes, along with any errors. The page also provides a link to view successfully generated code.

The generated Apex classes include stub and type classes for calling the third-party web service represented by the WSDL document. These classes allow you to call the external web service from Apex. For each generated class, a second class is created with the same name and the prefix `Async`. The `calculatorServices` class is for synchronous callouts. The `AsyncCalculatorServices` class is for asynchronous callouts.

Execute the Callout

Prerequisites

Before you run this example, authorize the endpoint URL of the web service callout, https://th-apex-soap-service.herokuapp.com, using the steps from the Authorize Endpoint Addresses section.

Now you can execute the callout and see if it correctly adds two numbers. Have a calculator handy to check the results.

1. Open the Developer Console under Your Name or the quick access menu ( ).
2. In the Developer Console, select .
3. Delete all existing code and insert the following snippet.

```
calculatorServices.CalculatorImplPort calculator = new
calculatorServices.CalculatorImplPort();
Double x = 1.0;
Double y = 2.0;
Double result = calculator.doAdd(x,y);
System.debug(result);
```

4. Select **Open Log**, and then click **Execute**.
5. After the debug log opens, click **Debug Only** to view the output of the `System.debug` statements. The log should display `3.0`.

Test Web Service Callouts

All experienced Apex developers know that to deploy or package Apex code, at least 75% of that code must have test coverage. This coverage includes our classes generated by WSDL2Apex. You might have heard this before, but test methods don't support web service callouts, and tests that perform web service callouts fail. So, we have a little work to do. To prevent tests from failing and to increase code coverage, Apex provides a built-in `WebServiceMock` interface and the `Test.setMock` method. You can use this interface to receive fake responses in a test method, thereby providing the necessary test coverage.

Specify a Mock Response for Callouts

When you create an Apex class from a WSDL, the methods in the autogenerated class call `WebServiceCallout.invoke`, which performs the callout to the external service. When testing these methods, you can instruct the Apex runtime to generate a fake response whenever `WebServiceCallout.invoke` is called. To do so, implement the `WebServiceMock` interface and specify a fake response for the testing runtime to send.

Instruct the Apex runtime to send this fake response by calling `Test.setMock` in your test method. For the first argument, pass `WebServiceMock.class`. For the second argument, pass a new instance of your `WebServiceMock` interface implementation.

```
Test.setMock(WebServiceMock.class, new
MyWebServiceMockImpl());
```

That's a lot to grok, so let's look at some code for a complete example. In this example, you create the class that makes the callout, a mock implementation for testing, and the test class itself.

1. In the Developer Console, select .
2. For the class name, enter `AwesomeCalculator` and then click **OK**.
3. Replace autogenerated code with the following class definition.

```
public class AwesomeCalculator {
    public static Double add(Double x, Double y) {
        calculatorServices.CalculatorImplPort calculator
=
            new calculatorServices.CalculatorImplPort();
        return calculator.doAdd(x,y);
    }
}
```

4. Press **CTRL+S** to save.

   Create your mock implementation to fake the callout during testing. Your implementation of `WebServiceMock` calls the `doInvoke` method, which returns the response you specify for testing. Most of this code is boilerplate. The hardest part of this exercise is figuring out how the web service returns a response so that you can fake a value.

5. In the Developer Console, select .
6. For the class name, enter `CalculatorCalloutMock` and then click **OK**.
7. Replace the autogenerated code with the following class definition.

```
@isTest
global class CalculatorCalloutMock implements WebServiceMock
{
   global void doInvoke(
           Object stub,
           Object request,
           Map<String, Object> response,
           String endpoint,
           String soapAction,
           String requestName,
           String responseNS,
           String responseName,
           String responseType) {
       // start - specify the response you want to send
       calculatorServices.doAddResponse response_x =
           new calculatorServices.doAddResponse();
       response_x.return_x = 3.0;
       // end
       response.put('response_x', response_x);
   }
}
```

8. Press **CTRL+S** to save.

   Lastly, your test method needs to instruct the Apex runtime to send the fake response by calling `Test.setMock` before making the callout in the `AwesomeCalculator` class. Like any other test method, we assert that the correct result from our mock response was received.

9. In the Developer Console, select .
10. For the class name, enter `AwesomeCalculatorTest` and then click **OK**.
11. Replace the autogenerated code with the following class definition.

```
@isTest
private class AwesomeCalculatorTest {
    @isTest static void testCallout() {
        // This causes a fake response to be generated
        Test.setMock(WebServiceMock.class, new
CalculatorCalloutMock());
        // Call the method that invokes a callout
        Double x = 1.0;
        Double y = 2.0;
        Double result = AwesomeCalculator.add(x, y);
        // Verify that a fake result is returned
        System.assertEquals(3.0, result);
    }
}
```

12. Press **CTRL+S** to save.
13. To run the test, select .

The `AwesomeCalculator` class should now display 100% code coverage!

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Apex Web Services

Learning Objectives

After completing this module, you'll be able to:

- Describe the two types of Apex web services and provide a high-level overview of these services.
- Create an Apex REST class that contains methods for each HTTP method.
- Invoke a custom Apex REST method with an endpoint.

- Pass data to a custom Apex REST method by sending a request body in JSON format.
- Write a test method for an Apex REST method and set properties in a test REST request.
- Write a test method for an Apex REST method by calling the method with parameter values.

Expose Your Apex Class as a Web Service

You can expose your Apex class methods as a REST or SOAP web service operation. By making your methods callable through the web, your external applications can integrate with Salesforce to perform all sorts of nifty operations.

For example, say your company's call center is using an internal application to manage on-premises resources. Customer support representatives are expected to use the same application to perform their daily work, including managing case records in Salesforce. By using one interface, representatives can view and update case records and access internal resources. The application calls an Apex web service class to manage Salesforce case records.

Expose a Class as a REST Service

Making your Apex class available as a REST web service is straightforward. Define your class as global, and define methods as global static. Add annotations to the class and methods. For example, this sample Apex REST class uses one method. The `getRecord` method is a custom REST API call. It's annotated with `@HttpGet` and is invoked for a GET request.

```
@RestResource(urlMapping='/Account/*')
global with sharing class MyRestResource
{
    @HttpGet
    global static Account getRecord() {
        // Add your code
    }
}
```

As you can see, the class is annotated with `@RestResource(urlMapping='/Account/*)`. The base endpoint for Apex REST is https://*yourInstance*.salesforce.com/services/apexrest/. The URL mapping is appended to the base endpoint to form the endpoint for your REST service. For example, in the class example, the REST endpoint is https://*yourInstance*.salesforce.com/services/apexrest/Account/. For your org, it could look something like, https://*yourInstance*.salesforce.com/services/apexrest/Account/.

The URL mapping is case-sensitive and can contain a wildcard character (*).

Define each exposed method as `global static` and add an annotation to associate it with an HTTP method. The following annotations are available. You can use each annotation only once in each Apex class.

Annotation	Action	Details
`@HttpGet`	Read	Reads or retrieves records.
`@HttpPost`	Create	Creates records.
`@HttpDelete`	Delete	Deletes records.
`@HttpPut`	Upsert	Typically used to update existing records or create records.
`@HttpPatch`	Update	Typically used to update fields in existing records.

Expose a Class as a SOAP Service

Making your Apex class available as a SOAP web service is as easy as with REST. Define your class as global. Add the `webservice` keyword and the `static` definition modifier to each method you want to expose. The `webservice` keyword provides global access to the method it is added to.

For example, here's a sample class with one method. The `getRecord` method is a custom SOAP API call that returns an Account record.

```
global with sharing class MySOAPWebService {
    webservice static Account getRecord(String id)
{
        // Add your code
    }
}
```

The external application can call your custom Apex methods as web service operations by consuming the class WSDL file. Generate this WSDL for your class from the class detail page, accessed from the Apex Classes page in Setup. You typically send the WSDL file to third-party developers (or use it yourself) to write integrations for your web service.

Because platform security is a first-class Salesforce citizen, your web service requires authentication. In addition to the Apex class WSDL, external applications must use either the Enterprise WSDL or the Partner WSDL for login functionality.

Apex REST Walkthrough

Now the fun stuff. The next few steps walk you through the process of building an Apex REST service. First, you create the Apex class that is exposed as a REST service. Then you try calling a few methods from a client, and finally write unit tests. There's quite a bit of code, but it will be worth the effort!

Your Apex class manages case records. The class contains five methods, and each method corresponds to an HTTP method. For example, when the client application invokes a REST call for the GET HTTP method, the `getCaseById` method is invoked.

Because the class is defined with a URL mapping of /Cases/*, the endpoint used to call this REST service is any URI that starts with https://*yourInstance*.salesforce.com/services/apexrest/Cases/.

We suggest that you also think about versioning your API endpoints so that you can provide upgrades in functionality without breaking existing code. You could create two classes specifying URL mappings of /Cases/v1/* and /Cases/v2/* to implement this functionality.

Let's get started by creating an Apex REST class.

1. Open the Developer Console under Your Name or the quick access menu ( ).
2. In the Developer Console, select .
3. For the class name, enter `CaseManager` and then click **OK**.
4. Replace the autogenerated code with the following class definition.

```
@RestResource(urlMapping='/Cases/*')
global with sharing class CaseManager {

    @HttpGet
    global static Case getCaseById() {
        RestRequest request = RestContext.request;
        // grab the caseId from the end of the URL
        String caseId = request.requestURI.substring(
          request.requestURI.lastIndexOf('/')+1);
        Case result =  [SELECT CaseNumber,Subject,Status,Origin,Priority
                        FROM Case
                        WHERE Id = :caseId];
        return result;
    }

    @HttpPost
    global static ID createCase(String subject, String status,
        String origin, String priority) {
        Case thisCase = new Case(
            Subject=subject,
            Status=status,
            Origin=origin,
            Priority=priority);
        insert thisCase;
        return thisCase.Id;
    }

    @HttpDelete
    global static void deleteCase() {
        RestRequest request = RestContext.request;
        String caseId = request.requestURI.substring(
            request.requestURI.lastIndexOf('/')+1);
        Case thisCase = [SELECT Id FROM Case WHERE Id = :caseId];
        delete thisCase;
    }

    @HttpPut
    global static ID upsertCase(String subject, String status,
        String origin, String priority, String id) {
        Case thisCase = new Case(
                Id=id,
                Subject=subject,
                Status=status,
                Origin=origin,
                Priority=priority);
        // Match case by Id, if present.
        // Otherwise, create new case.
        upsert thisCase;
        // Return the case ID.
        return thisCase.Id;
    }

    @HttpPatch
    global static ID updateCaseFields() {
        RestRequest request = RestContext.request;
        String caseId = request.requestURI.substring(
            request.requestURI.lastIndexOf('/')+1);
        Case thisCase = [SELECT Id FROM Case WHERE Id = :caseId];
        // Deserialize the JSON string into name-value pairs
        Map<String, Object> params = (Map<String,
Object>)JSON.deserializeUntyped(request.requestbody.tostring());
        // Iterate through each parameter field and value
        for(String fieldName : params.keySet()) {
            // Set the field and value on the Case sObject
            thisCase.put(fieldName, params.get(fieldName));
        }
        update thisCase;
        return thisCase.Id;
    }

}
```

5. Press **CTRL+S** to save.

Create a Record with a POST Method

Let's use the Apex REST class that you've just created and have some fun. First, we'll call the POST method to create a case record.

To invoke your REST service, you need to use a… REST client! You can use almost any REST client, such as your own API client, the cURL command-line tool, or the curl library for PHP. We'll use the Workbench tool as our REST client application, but we'll take a peek at cURL later on.

Apex REST supports two formats for representations of resources: JSON and XML. JSON representations are passed by default in the body of a request or response, and the format is indicated by the `Content-Type` property in the HTTP header. Since JSON is easier to read and understand than XML, this unit uses JSON exclusively. In this step, you send a case record in JSON format.

Apex REST supports OAuth 2.0 and session authentication mechanisms. In simple terms, this means that we use industry standards to keep your application and data safe. Fortunately, you can use Workbench to make testing easier. Workbench is a powerful, web-based suite of tools for administrators and developers to interact with orgs via Force.com APIs. With Workbench, you use session authentication as you log in with your username and password to Salesforce. And you use the REST Explorer to call your REST service.

1. Navigate to https://workbench.developerforce.com/login.php.
2. For Environment, select **Production**.
3. Select the latest API version from the **API Version** drop-down.
4. Accept the terms of service, and click **Login with Salesforce**.
5. To allow Workbench to access your information, click **Allow**.
6. Enter your login credentials and then click **Log in to Salesforce**.
7. After logging in, select .
8. Select **POST**.
9. The URL path that REST Explorer accepts is relative to the instance URL of your org. Provide only the path that is appended to the instance URL. In the relative URI input field, replace the default URI with `/services/apexrest/Cases/`.
10. For the request body, insert the following JSON string representation of the object to insert.

```
{
  "subject" : "Bigfoot
Sighting!",
  "status" : "New",
  "origin" : "Phone",
  "priority" : "Low"
}
```

11. Click **Execute**.

    This invocation calls the method that is associated with the POST HTTP method, namely the `createCase` method.

12. To view the response returned, click **Show Raw Response**.
    The returned response looks similar to this response. The response contains the ID of the new case record. Your ID value is likely different from `50061000000t7kYAAQ`. Save your ID value to use in the next steps.

```
HTTP/1.1 200 OK
Date: Wed, 07 Oct 2015 14:18:20 GMT
Set-Cookie: BrowserId=F1wxIhHPQHCXp6wrvqToXA;Path=/;Domain=.salesforce.com;Expires=Sun, 06-Dec-2015
14:18:20 GMT
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: application/json;charset=UTF-8
Content-Encoding: gzip
Transfer-Encoding: chunked

"50061000000t7kYAAQ"
```

Retrieve Data with a Custom GET Method

By following similar steps as before, use Workbench to invoke the GET HTTP method.

1. In Workbench, select **GET**.
2. Enter the URI /services/apexrest/Cases/, replacing with the ID of the record you created in the previous step.
3. Click **Execute**.

   This invocation calls the method associated with the GET HTTP method, namely the `getCaseById` method.

4. To view the response returned, click **Show Raw Response**.

   The returned response looks similar to this response. The response contains the fields that the method queried for the new case record.

```
HTTP/1.1 200 OK
Date: Wed, 07 Oct 2015 14:28:20 GMT
Set-Cookie: BrowserId=j5qAnPDdRxSu8eHGqaRVLQ;Path=/;Domain=.salesforce.com;Expires=Sun, 06-Dec-2015
14:28:20 GMT
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: application/json;charset=UTF-8
Content-Encoding: gzip
Transfer-Encoding: chunked

{
  "attributes" : {
    "type" : "Case",
    "url" : "/services/data/v34.0/sobjects/Case/50061000000t7kYAAQ"
  },
  "CaseNumber" : "00001026",
  "Subject" : "Bigfoot Sighting!",
  "Status" : "New",
  "Origin" : "Phone",
  "Priority" : "Low",
  "Id" : "50061000000t7kYAAQ"
}
```

Retrieve Data Using cURL

Every good developer should know at least three things: 1) how to make an animated GIF of yourself eating ice cream; 2) the value of pi to 25 decimal places; and 3) how to use cURL. The first two are beyond the scope of this module, so we'll concentrate on the last one.

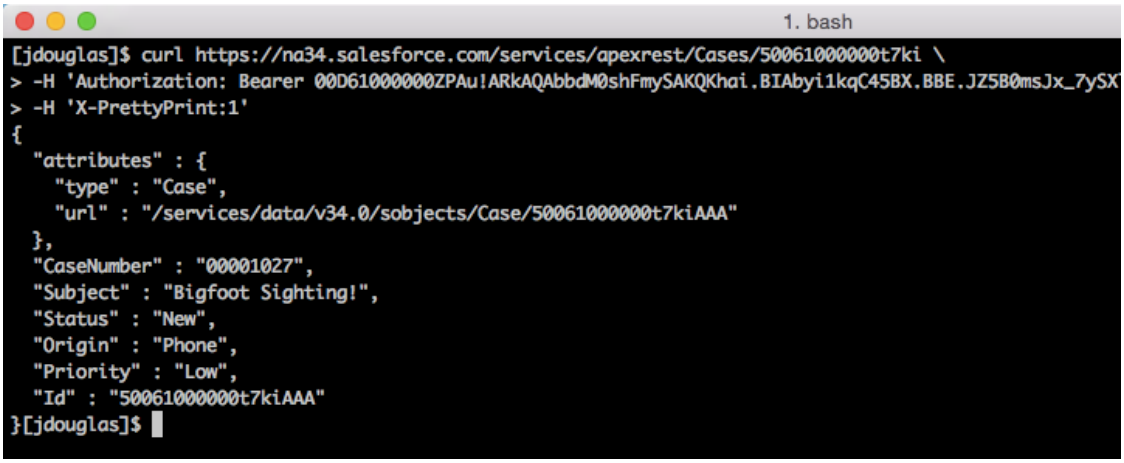
cURL is a command-line tool for getting or sending files using URL syntax. It comes in quite handy when working with REST endpoints. Instead of using Workbench for your Apex REST service, you'll use cURL to invoke the GET HTTP method. Each time you "cURL" your REST endpoint, you pass along the session ID for authorization. You were spoiled when working in Workbench because it passes the session ID for you, under the covers, after you log in.

You can obtain a session ID a few different ways, but the easiest—because you're already in Workbench—is to grab the ID from there. In the top menu, select and then expand the Connection folder to find the session ID.

Now open up terminal or the command prompt and enter your cURL command, which will be similar to the following.

```
curl https://yourInstance.salesforce.com/services/apexrest/Cases/<Record_ID> -H 'Authorization: Bearer
<your_session_id>' -H 'X-PrettyPrint:1'
```

After pressing Enter, you see something similar to the following. Now that you are a command-line master, feel free to cURL, jq, sed, awk, and grep to your heart's content. For more info on cURL, see the Resources section.



Update Data with a Custom PUT or PATCH Method

You can update records with the PUT or PATCH HTTP methods. The PUT method either updates the entire resource, if it exists, or creates the resource if it doesn't exist. PUT is essentially an upsert method. The PATCH method updates only the specified portions of an existing resource. In Apex, update

operations update only the specified fields and don't overwrite the entire record. We'll write some Apex code to determine whether our methods update or upsert.

Update Data with the PUT Method

The `upsertCase` method that you added to the `CaseManager` class implements the PUT action. This method is included here for your reference. The method uses the built-in `upsert` Apex DML method to either create or overwrite case record fields by matching the ID value. If an ID is sent in the body of the request, the case sObject is populated with it. Otherwise, the case sObject is created without an ID. The `upsert` method is invoked with the populated case sObject, and the DML statement does the rest. Voila!

```
@HttpPut
global static ID upsertCase(String subject, String
status,
    String origin, String priority, String id) {
    Case thisCase = new Case(
        Id=id,
        Subject=subject,
        Status=status,
        Origin=origin,
        Priority=priority);
    // Match case by Id, if present.
    // Otherwise, create new case.
    upsert thisCase;
    // Return the case ID.
    return thisCase.Id;
}
```

To invoke the PUT method:

1. In Workbench REST Explorer, select **PUT**.
2. For the URI, enter `/services/apexrest/Cases/`.
3. The `upsertCase` method expects the field values to be passed in the request body. Add the following for the request body, and then replace with the ID of the case record you created earlier.

   ```
   {
     "id": "<Record_ID>",
     "status" : "Working",
     "subject" : "Bigfoot
   Sighting!",
     "priority" : "Medium"
   }
   ```

   Note

   The ID field is optional. To create a case record, omit this field. In our example, you're passing this field because you want to update the case record.

4. Click **Execute**.

   This request invokes the `upsertCase` method from your REST service. The Status, Subject, and Priority fields are updated. The subject is updated, even though its value matches the old subject. Also, because the request body didn't contain a value for the Case Origin field, the origin parameter in the `upsertCase` method is null. As a result, when the record is updated, the Origin field is cleared.

   To check these fields, view this record in Salesforce by navigating to https://*yourInstance*.salesforce.com/.

Update Data with the PATCH Method

As an alternative to the PUT method, use the PATCH method to update record fields. You can implement the PATCH method in different ways. One way is to specify parameters in the method for each field to update. For example, you can create a method to update the priority of a case with this signature: `updateCasePriority(String priority)`. To update multiple fields, you can list all the desired fields as parameters.

Another approach that provides more flexibility is to pass the fields as JSON name/value pairs in the request body. That way the method can accept an arbitrary number of parameters, and the parameters aren't fixed in the method's signature. Another advantage of this approach is that no field is accidentally cleared because of being null. The `updateCaseFields` method that you added to the `CaseManager` class uses this second approach. This method deserialize the JSON string from the request body into a map of name/value pairs and uses the sObject `PUT` method to set the fields.

```
@HttpPatch
global static ID updateCaseFields() {
    RestRequest request = RestContext.request;
    String caseId = request.requestURI.substring(
        request.requestURI.lastIndexOf('/')+1);
    Case thisCase = [SELECT Id FROM Case WHERE Id = :caseId];
    // Deserialize the JSON string into name-value pairs
    Map<String, Object> params = (Map<String,
Object>)JSON.deserializeUntyped(request.requestbody.tostring());
    // Iterate through each parameter field and value
    for(String fieldName : params.keySet()) {
        // Set the field and value on the Case sObject
        thisCase.put(fieldName, params.get(fieldName));
    }
    update thisCase;
    return thisCase.Id;
}
```

To invoke the PATCH method:

1. In Workbench REST Explorer, click **PATCH**.
2. For the URI, enter `/services/apexrest/Cases/`. Replace with the ID of the case record created earlier. Enter the following JSON in the Request Body.

   ```
   {
     "status" :
   "Escalated",
     "priority" : "High"
   }
   ```

   This JSON has two field values: status and priority. The `updateCaseFields` method retrieves these values from the submitted JSON and are used to specify the fields to update in the object.

3. Click **Execute**.

   This request invokes the `updateCaseFields` method in your REST service. The Status and Priority fields of the case record are updated to new values. To check these fields, view this record in Salesforce by navigating to https://*yourInstance*.salesforce.com/.

Test Your Apex REST Class

Testing your Apex REST class is similar to testing any other Apex class—just call the class methods by passing in parameter values and then verify the results. For methods that don't take parameters or that rely on information in the REST request, create a test REST request.

In general, here's how you test Apex REST services. To simulate a REST request, create a `RestRequest` in the test method, and then set properties on the request as follows. You can also add params that you "pass" in the request to simulate URI parameters.

```
// Set up a test request
RestRequest request = new RestRequest();

// Set request properties
request.requestUri =

'https://yourInstance.salesforce.com/services/apexrest/Cases/'
    + recordId;
request.httpMethod = 'GET';

// Set other properties, such as parameters
request.params.put('status', 'Working');

// more awesome code here....
// Finally, assign the request to RestContext if used
RestContext.request = request;
```

If the method you're testing accesses request values through `RestContext`, assign the request to `RestContext` to populate it (`RestContext.request = request;`).

Now, let's save the entire class in the Developer Console and run the results.

1. In the Developer Console, select .

2. For the class name, enter `CaseManagerTest` and then click **OK**.
3. Replace the autogenerated code with the following class definition.

```
@IsTest
private class CaseManagerTest {

    @isTest static void testGetCaseById() {
        Id recordId = createTestRecord();
        // Set up a test request
        RestRequest request = new RestRequest();
        request.requestUri =

'https://yourInstance.salesforce.com/services/apexrest/Cases/'
            + recordId;
        request.httpMethod = 'GET';
        RestContext.request = request;
        // Call the method to test
        Case thisCase = CaseManager.getCaseById();
        // Verify results
        System.assert(thisCase != null);
        System.assertEquals('Test record', thisCase.Subject);
    }

    @isTest static void testCreateCase() {
        // Call the method to test
        ID thisCaseId = CaseManager.createCase(
            'Ferocious chipmunk', 'New', 'Phone', 'Low');
        // Verify results
        System.assert(thisCaseId != null);
        Case thisCase = [SELECT Id,Subject FROM Case WHERE
Id=:thisCaseId];
        System.assert(thisCase != null);
        System.assertEquals(thisCase.Subject, 'Ferocious chipmunk');
    }

    @isTest static void testDeleteCase() {
        Id recordId = createTestRecord();
        // Set up a test request
        RestRequest request = new RestRequest();
        request.requestUri =

'https://yourInstance.salesforce.com/services/apexrest/Cases/'
            + recordId;
        request.httpMethod = 'GET';
        RestContext.request = request;
        // Call the method to test
        CaseManager.deleteCase();
        // Verify record is deleted
        List<Case> cases = [SELECT Id FROM Case WHERE Id=:recordId];
        System.assert(cases.size() == 0);
    }

    @isTest static void testUpsertCase() {
        // 1. Insert new record
        ID case1Id = CaseManager.upsertCase(
                'Ferocious chipmunk', 'New', 'Phone', 'Low', null);
        // Verify new record was created
        System.assert(Case1Id != null);
        Case case1 = [SELECT Id,Subject FROM Case WHERE Id=:case1Id];
        System.assert(case1 != null);
        System.assertEquals(case1.Subject, 'Ferocious chipmunk');
        // 2. Update status of existing record to Working
        ID case2Id = CaseManager.upsertCase(
                'Ferocious chipmunk', 'Working', 'Phone', 'Low',
case1Id);
        // Verify record was updated
        System.assertEquals(case1Id, case2Id);
        Case case2 = [SELECT Id,Status FROM Case WHERE Id=:case2Id];
        System.assert(case2 != null);
        System.assertEquals(case2.Status, 'Working');
    }

    @isTest static void testUpdateCaseFields() {
        Id recordId = createTestRecord();
```

```
        RestRequest request = new RestRequest();
        request.requestUri =

'https://yourInstance.salesforce.com/services/apexrest/Cases/'
            + recordId;
        request.httpMethod = 'PATCH';
        request.addHeader('Content-Type', 'application/json');
        request.requestBody = Blob.valueOf('{"status": "Working"}');
        RestContext.request = request;
        // Update status of existing record to Working
        ID thisCaseId = CaseManager.updateCaseFields();
        // Verify record was updated
        System.assert(thisCaseId != null);
        Case thisCase = [SELECT Id,Status FROM Case WHERE
Id=:thisCaseId];
        System.assert(thisCase != null);
        System.assertEquals(thisCase.Status, 'Working');
    }

    // Helper method
    static Id createTestRecord() {
        // Create test record
        Case caseTest = new Case(
            Subject='Test record',
            Status='New',
            Origin='Phone',
            Priority='Medium');
        insert caseTest;
        return caseTest.Id;
    }

}
```

4. Press `CTRL+S` to save.
5. Run all the tests in your org by selecting .

The test results display in the Tests tab. After the test execution finishes, check the CaseManager row in the Overall Code Coverage pane. It's at 100% coverage.

Tell Me More ...

Learn about supported data types and namespaces in Apex REST, Salesforce APIs, and security considerations.

Supported Data Types for Apex REST

Apex REST supports these data types for parameters and return values.

- Apex primitives (excluding sObject and Blob).
- sObjects
- Lists or maps of Apex primitives or sObjects (only maps with String keys are supported).
- User-defined types that contain member variables of the types listed above.

Namespaces in Apex REST Endpoints

Apex REST methods can be used in managed and unmanaged packages. When calling Apex REST methods that are contained in a managed package, you need to include the managed package namespace in the REST call URL. For example, if the class is contained in a managed package namespace called `packageNamespace` and the Apex REST methods use a URL mapping of /MyMethod/*, the URL used via REST to call these methods would be of the form https://*instance*.salesforce.com/services/apexrest/packageNamespace/MyMethod/.

Custom Apex Web Services and Salesforce APIs

Instead of using custom Apex code for REST and SOAP services, external applications can integrate with Salesforce by using Salesforce's REST and SOAP APIs. These APIs let you create, update, and delete records. However, the advantage of using Apex web services is that Apex methods can encapsulate complex logic. This logic is hidden from the consuming application. Also, the Apex class operations can be faster than making individual API calls, because fewer roundtrips are performed between the client and the Salesforce servers. With an Apex web service call, there is only one request sent, and all operations within the method are performed on the server.

Security Considerations for Apex Web Services

The security context under which Apex web service methods run differs from the security context of Salesforce APIs. Unlike Salesforce APIs, Apex web service methods run with system privileges and don't respect the user's object and field permissions. However, Apex web service methods enforce sharing rules when declared with the `with sharing` keyword.

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Introducing Asynchronous Processing

Learning Objectives

After completing this unit, you'll be able to:

- Explain the difference between synchronous and asynchronous processing.
- Choose which kind of asynchronous Apex to use in various scenarios.

Asynchronous Apex

In a nutshell, asynchronous Apex is used to run processes in a separate thread, at a later time.

An asynchronous process is a process or function that executes a task "in the background" without the user having to wait for the task to finish.

Here's a real-world example. Let's say you have a list of things to accomplish before your weekly Dance Dance Revolution practice. Your car is making a funny noise, you need a different color hair gel and you have to pick up your uniform from your mom's house. You could take your car to the mechanic and wait until it is fixed before completing the rest of your list (synchronous processing), or you could leave it there and get your other things done, and have the shop call you when it's fixed (asynchronous processing). If you want to be home in time to iron your spandex before practice, asynchronous processing allows you to get more stuff done in the same amount of time without the needless waiting.

You'll typically use Asynchronous Apex for callouts to external systems, operations that require higher limits, and code that needs to run at a certain time. The key benefits of asynchronous processing include:

User efficiency

> Let's say you have a process that makes many calculations on a custom object whenever an Opportunity is created. The time needed to execute these calculations could range from a minor annoyance to a productivity blocker for the user. Since these calculations don't affect what the user is currently doing, making them wait for a long running process is not an efficient use of their time. With asynchronous processing the user can get on with their work, the processing can be done in the background and the user can see the results at their convenience.

Scalability

> By allowing some features of the platform to execute when resources become available at some point in the future, resources can be managed and scaled quickly. This allows the platform to handle more jobs using parallel processing.

Higher Limits

> Asynchronous processes are started in a new thread, with higher governor and execution limits. And to be honest, doesn't everyone want higher governor and execution limits?

Asynchronous Apex comes in a number of different flavors. We'll get into more detail for each one shortly, but here's a high level overview.

Type	Overview	Common Scenarios
Future Methods	Run in their own thread, and do not start until resources are available.	Web service callout.
Batch Apex	Run large jobs that would exceed normal processing limits.	Data cleansing or archiving of records.
Queueable Apex	Similar to future methods, but provide additional job chaining and allow more complex data types to be used.	Performing sequential processing operations with external Web services.
Scheduled Apex	Schedule Apex to run at a specified time.	Daily or weekly tasks.

It's also worth noting that these different types of asynchronous operations are not mutually exclusive. For instance, a common pattern is to kick off a Batch Apex job from a Scheduled Apex job.

Increased Governor and Execution Limits

One of the main benefits of running asynchronous Apex is higher governor and execution limits. For example, the number of SOQL queries is doubled from 100 to 200 queries when using asynchronous calls. The total heap size and maximum CPU time are similarly larger for asynchronous calls.

Not only do you get higher limits with async, but also those governor limits are independent of the limits in the synchronous request that queued the async request initially. That's a mouthful, but essentially, you have two separate Apex invocations, and more than double the processing capability. This comes in handy for instances when you want to do as much processing as you can in the current transaction but when you start to get close to governor limits, continue asynchronously.

If you enjoy reading about heap sizes, maximum execution times and limits in general, see Execution Governors and Limits for compelling details.

How Asynchronous Processing Works

Asynchronous processing, in a multi-tenant environment, presents some challenges:

Ensure fairness of processing

> Make sure every customer gets a fair share of processing resources.

Ensure fault tolerance

> Make sure no asynchronous requests are lost due to equipment or software failures.

The platform uses a queue-based asynchronous processing framework. This framework is used to manage asynchronous requests for multiple organizations within each instance. The request lifecycle is made up of three parts:

Enqueue

> The request gets put into the queue. This could be an Apex batch request, future Apex request or one of many others. The platform will enqueue requests along with the appropriate data to process that request.

Persistence

> The enqueued request is persisted. Requests are stored in persistent storage for failure recovery and to provide transactional capabilities.

Dequeue

> The enqueued request is removed from the queue and processed. Transaction management occurs in this step to assure messages are not lost if there is a processing failure.

Each request is processed by a handler. The handler is the code that performs functions for a specific request type. Handlers are executed by a finite number worker threads on each of the application servers that make up an instance. The threads request work from the queuing framework and when received, start a specific handler to do the work.

Resource Conservation

Asynchronous processing has lower priority than real-time interaction via the browser and API. To ensure there are sufficient resources to handle an increase in computing resources, the queuing framework monitors system resources such as server memory and CPU usage and reduce asynchronous processing when thresholds are exceeded. This is a fancy way of saying that the multitenant system protects itself. If an org tries to "gobble up" more than its share of resources, asynchronous processing is suspended until a normal threshold is reached. The long and short of it is that there's no guarantee on processing time, but it'll all work out in the end.

Using Future Methods

Learning Objectives

After completing this unit, you'll know:

- When to use future methods.
- The limitations of using future methods.
- How to use future methods for callouts.
- Future method best practices.

Future Apex

Future Apex is used to run processes in a separate thread, at a later time when system resources become available.

Note: Technically, you use the `@future` annotation to identify methods that run asynchronously. However, because "methods identified with the `@future` annotation" is laborious, they are commonly referred to as "future methods" and that's how we'll reference them for the remainder of this module.

When using synchronous processing, all method calls are made from the same thread that is executing the Apex code, and no additional processing can

occur until the process is complete. You can use future methods for any operation you'd like to run asynchronously in its own thread. This provides the benefits of not blocking the user from performing other operations and providing higher governor and execution limits for the process. Everyone's a winner with asynchronous processing.

Future methods are typically used for:

- Callouts to external Web services. If you are making callouts from a trigger or after performing a DML operation, you must use a future or queueable method. A callout in a trigger would hold the database connection open for the lifetime of the callout and that is a "no-no" in a multitenant environment.
- Operations you want to run in their own thread, when time permits such as some sort of resource-intensive calculation or processing of records.
- Isolating DML operations on different sObject types to prevent the mixed DML error. This is somewhat of an edge-case but you may occasionally run across this issue. See sObjects That Cannot Be Used Together in DML Operations for more details.

Future Method Syntax

Future methods must be static methods, and can only return a void type. The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types. Notably, future methods can't take standard or custom objects as arguments. A common pattern is to pass the method a List of record IDs that you want to process asynchronously.

```
global class SomeClass {
  @future
  public static void someFutureMethod(List<Id> recordIds) {
    List<Account> accounts = [Select Id, Name from Account Where Id IN
:recordIds];
    // process account records to do awesome stuff
  }
}
```

Note

The reason why objects can't be passed as arguments to future methods is because the object can change between the time you call the method and the time that it actually executes. Remember, future methods are executed when system resources become available. In this case, the future method may have an old object value when it actually executes, which can cause all sorts of bad things to happen.

It's important to note that future methods are not guaranteed to execute in the same order as they are called. Again, future methods are not guaranteed to execute in the same order as they are called. If you need this type of functionality then Queueable Apex might be a better solution. When using future methods, it's also possible that two future methods could run concurrently, which could result in record locking and a nasty runtime error if the two methods were updating the same record.

Sample Callout Code

To make a Web service callout to an external service or API, you create an Apex class with a future method that is marked with `(callout=true)`. The class below has methods for making the callout both synchronously and asynchronously where callouts are not permitted. We insert a record into a custom log object to track the status of the callout simply because logging is always fun to do!

```
public class SMSUtils {

    // Call async from triggers, etc, where callouts are not permitted.
    @future(callout=true)
    public static void sendSMSAsync(String fromNbr, String toNbr, String m)
{
        String results = sendSMS(fromNbr, toNbr, m);
        System.debug(results);
    }

    // Call from controllers, etc, for immediate processing
    public static String sendSMS(String fromNbr, String toNbr, String m) {
        // Calling 'send' will result in a callout
        String results = SmsMessage.send(fromNbr, toNbr, m);
        insert new SMS_Log__c(to__c=toNbr, from__c=fromNbr,
msg__c=results);
        return results;
    }

}
```

Test Classes

Testing future methods is a little different than typical Apex testing. To test future methods, enclose your test code between the `startTest` and `stopTest` test methods. The system collects all asynchronous calls made after the startTest. When `stopTest` is executed, all these collected asynchronous processes are then run synchronously. You can then assert that the asynchronous call operated properly.

Note

Test code cannot actually send callouts to external systems, so you'll have to 'mock' the callout for test coverage. Check out the Apex Integration Services module for complete details on mocking callouts for testing.

Here's our mock callout class used for testing. The Apex testing framework utilizes this 'mock' response instead of making the actual callout to the REST API endpoint.

```
@isTest
global class SMSCalloutMock implements HttpCalloutMock {
    global HttpResponse respond(HttpRequest req) {
        // Create a fake response
        HttpResponse res = new HttpResponse();
        res.setHeader('Content-Type',
'application/json');
        res.setBody('{"status":"success"}');
        res.setStatusCode(200);
        return res;
    }
}
```

The test class contains a single test method, which tests both the asynchronous and synchronous methods as the former calls the latter.

```
@IsTest
private class Test_SMSUtils {

  @IsTest
  private static void testSendSms() {
    Test.setMock(HttpCalloutMock.class, new
SMSCalloutMock());
    Test.startTest();
      SMSUtils.sendSMSAsync('111', '222', 'Greetings!');
    Test.stopTest();
    // runs callout and check results
    List<SMS_Log__c> logs = [select msg__c from SMS_Log__c];
    System.assertEquals(1, logs.size());
    System.assertEquals('success', logs[0].msg__c);
  }

}
```

Best Practices

Since every future method invocation adds one request to the asynchronous queue, avoid design patterns that add large numbers of future requests over a short period of time. If your design has the potential to add 2000 or more requests at a time, requests could get delayed due to flow control. Here are some best practices you want to keep in mind:

- Ensure that future methods execute as fast as possible.
- If using Web service callouts, try to bundle all callouts together from the same future method, rather than using a separate future method for each callout.
- Conduct thorough testing at scale. Test that a trigger enqueuing the `@future` calls is able to handle a trigger collection of 200 records. This helps determine if delays may occur given the design at current and future volumes.
- Consider using Batch Apex instead of future methods to process large number of records asynchronously. This is more efficient than creating a future request for each record.

Things to Remember

Future methods are a great tool, but with great power comes great responsibility. Here are some things to keep in mind when using them:

- Methods with the future annotation must be static methods, and can only return a `void` type.
- The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types; future methods can't take objects as arguments.
- Future methods won't necessarily execute in the same order they are called. In addition, it's possible that two future methods could run concurrently, which could result in record locking if the two methods were updating the same record.
- Future methods can't be used in Visualforce controllers in `getMethodName()`, `setMethodName()`, nor in the constructor.
- You can't call a future method from a future method. Nor can you invoke a trigger that calls a future method while running a future method. See the link in the Resources for preventing recursive future method calls.
- The `getContent()` and `getContentAsPDF()` methods can't be used in methods with the future annotation.
- You're limited to 50 future calls per Apex invocation, and there's an additional limit on the number of calls in a 24-hour period. For more information on limits, see the link below.

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Using Batch Apex

Learning Objectives

After completing this unit, you'll know:

- Where to use Batch Apex.
- The higher Apex limits when using batch.

- Batch Apex syntax.
- Batch Apex best practices.

Batch Apex

Batch Apex is used to run large jobs (think thousands or millions of records!) that would exceed normal processing limits. Using Batch Apex, you can process records asynchronously in batches (hence the name, “Batch Apex”) to stay within platform limits. If you have a lot of records to process, for example, data cleansing or archiving, Batch Apex is probably your best solution.

Here’s how Batch Apex works under the hood. Let’s say you want to process 1 million records using Batch Apex. The execution logic of the batch class is called once for each batch of records you are processing. Each time you invoke a batch class, the job is placed on the Apex job queue and is executed as a discrete transaction. This functionality has two awesome advantages:

- Every transaction starts with a new set of governor limits, making it easier to ensure that your code stays within the governor execution limits.
- If one batch fails to process successfully, all other successful batch transactions aren’t rolled back.

Batch Apex Syntax

To write a Batch Apex class, your class must implement the `Database.Batchable` interface and include the following three methods:

start

> Used to collect the records or objects to be passed to the interface method `execute` for processing. This method is called once at the beginning of a Batch Apex job and returns either a Database.QueryLocator object or an Iterable that contains the records or objects passed to the job.
>
> Most of the time a `QueryLocator` does the trick with a simple SOQL query to generate the scope of objects in the batch job. But if you need to do something crazy like loop through the results of an API call or pre-process records before being passed to the `execute` method, you might want to check out the Custom Iterators link in the Resources section.
>
> With the QueryLocator object, the governor limit for the total number of records retrieved by SOQL queries is bypassed and you can query up to 50 million records. However, with an Iterable, the governor limit for the total number of records retrieved by SOQL queries is still enforced.

execute

> Performs the actual processing for each chunk or “batch” of data passed to the method. The default batch size is 200 records. Batches of records are not guaranteed to execute in the order they are received from the `start` method.
>
> This method takes the following:
>
> - A reference to the `Database.BatchableContext` object.
> - A list of sObjects, such as `List`, or a list of parameterized types. If you are using a `Database.QueryLocator`, use the returned list.

finish

> Used to execute post-processing operations (for example, sending an email) and is called once after all batches are processed.

Here’s what the skeleton of a Batch Apex class looks like:

```
global class MyBatchClass implements Database.Batchable<sObject> {

    global (Database.QueryLocator | Iterable<sObject>) start(Database.BatchableContext bc)
{
        // collect the batches of records or objects to be passed to execute
    }

    global void execute(Database.BatchableContext bc, List<P> records){
        // process each batch of records
    }

    global void finish(Database.BatchableContext bc){
        // execute any post-processing operations
    }

}
```

Invoking a Batch Class

To invoke a batch class, simply instantiate it and then call `Database.executeBatch` with the instance:

```
MyBatchClass myBatchObject = new MyBatchClass();
Id batchId =
Database.executeBatch(myBatchObject);
```

You can also optionally pass a second `scope` parameter to specify the number of records that should be passed into the execute method for each batch. Pro tip: you might want to limit this batch size if you are running into governor limits.

```
Id batchId = Database.executeBatch(myBatchObject,
100);
```

Each batch Apex invocation creates an AsyncApexJob record so that you can track the job's progress. You can view the progress via SOQL or manage your job in the Apex Job Queue. We'll talk about the Job Queue shortly.

```
AsyncApexJob job = [SELECT Id, Status, JobItemsProcessed, TotalJobItems, NumberOfErrors FROM AsyncApexJob WHERE
ID = :batchId ];
```

Using State in Batch Apex

Batch Apex is typically stateless. Each execution of a batch Apex job is considered a discrete transaction. For example, a batch Apex job that contains 1,000 records and uses the default batch size is considered five transactions of 200 records each.

If you specify `Database.Stateful` in the class definition, you can maintain state across all transactions. When using `Database.Stateful`, only instance member variables retain their values between transactions. Maintaining state is useful for counting or summarizing records as they're processed. In our next example, we'll be updating contact records in our batch job and want to keep track of the total records affected so we can include it in the notification email.

Sample Batch Apex Code

Now that you know how to write a Batch Apex class, let's see a practical example. Let's say you have a business requirement that states that all contacts for companies in the USA must have their parent company's billing address as their mailing address. Unfortunately, users are entering new contacts without the correct addresses! Will users never learn?! Write a Batch Apex class that ensures that this requirement is enforced.

The following sample class finds all account records that are passed in by the `start()` method using a `QueryLocator` and updates the associated contacts with their account's mailing address. Finally, it sends off an email with the results of the bulk job and, since we are using `Database.Stateful` to track state, the number of records updated.

```
global class UpdateContactAddresses implements
    Database.Batchable<sObject>, Database.Stateful {

    // instance member to retain state across transactions
    global Integer recordsProcessed = 0;

    global Database.QueryLocator start(Database.BatchableContext bc) {
        return Database.getQueryLocator(
            'SELECT ID, BillingStreet, BillingCity, BillingState, ' +
            'BillingPostalCode, (SELECT ID, MailingStreet, MailingCity, ' +
            'MailingState, MailingPostalCode FROM Contacts) FROM Account '
+
            'Where BillingCountry = \'USA\''
        );
    }

    global void execute(Database.BatchableContext bc, List<Account> scope){
        // process each batch of records
        List<Contact> contacts = new List<Contact>();
        for (Account account : scope) {
            for (Contact contact : account.contacts) {
                contact.MailingStreet = account.BillingStreet;
                contact.MailingCity = account.BillingCity;
                contact.MailingState = account.BillingState;
                contact.MailingPostalCode = account.BillingPostalCode;
                // add contact to list to be updated
                contacts.add(contact);
                // increment the instance member counter
                recordsProcessed = recordsProcessed + 1;
            }
        }
        update contacts;
    }

    global void finish(Database.BatchableContext bc){
        System.debug(recordsProcessed + ' records processed. Shazam!');
        AsyncApexJob job = [SELECT Id, Status, NumberOfErrors,
            JobItemsProcessed,
            TotalJobItems, CreatedBy.Email
            FROM AsyncApexJob
            WHERE Id = :bc.getJobId()];
        // call some utility to send email
        EmailUtils.sendMessage(a, recordsProcessed);
    }

}
```

The code should be fairly straightforward but can be a little abstract in reality. Here's what's going on in more detail:

- The `start` method provides the collection of all records that the `execute` method will process in individual batches. It returns the list of records to be processed by calling `Database.getQueryLocator` with a SOQL query. In this case we are simply querying for all Account records with a Billing Country of 'USA'.
- Each batch of 200 records is passed in the second parameter of the `execute` method. The `execute` method sets each contact's mailing address to the accounts' billing address and increments `recordsProcessed` to track the number of records processed.
- When the job is complete, the `finish` method performs a query on the AsyncApexJob object (a table that lists information about batch jobs) to get the status of the job, the submitter's email address, and some other information. It then sends a notification email to the job submitter that includes the job info and number of contacts updated.

Testing Batch Apex

Since Apex development and testing go hand in hand, here's how we test the above batch class. In a nutshell, we insert some records, call the Batch Apex class and then assert that the records were updated properly with the correct address.

```
@isTest
private class UpdateContactAddressesTest {

    @testSetup
    static void setup() {
        List<Account> accounts = new List<Account>();
        List<Contact> contacts = new List<Contact>();
        // insert 10 accounts
        for (Integer i=0;i<10;i++) {
            accounts.add(new Account(name='Account '+i,
                billingcity='New York', billingcountry='USA'));
        }
        insert accounts;
        // find the account just inserted. add contact for each
        for (Account account : [select id from account]) {
            contacts.add(new Contact(firstname='first',
                lastname='last', accountId=account.id));
        }
        insert contacts;
    }

    static testmethod void test() {
        Test.startTest();
        UpdateContactAddresses uca = new UpdateContactAddresses();
        Id batchId = Database.executeBatch(uca);
        Test.stopTest();

        // after the testing stops, assert records were updated properly
        System.assertEquals(10, [select count() from contact where MailingCity = 'New
York']);
    }

}
```

The `setup` method inserts 10 account records with the billing city of ‘New York’ and the billing country of ‘USA’. Then for each account, it creates an associated contact record. This data is used by the batch class.

Note

Make sure that the number of records inserted is less than the batch size of 200 because test methods can execute only one batch total.

In the test method, the `UpdateContactAddresses` batch class is instantiated, invoked by calling `Database.executeBatch` and passing it the instance of the batch class.

The call to `Database.executeBatch` is included within the `Test.startTest` and `Test.stopTest` block. This is where all of the magic happens. The job executes after the call to `Test.stopTest`. Any asynchronous code included within `Test.startTest` and `Test.stopTest` is executed synchronously after `Test.stopTest`.

Finally, the test verifies that all contact records were updated correctly by checking that the number of contact records with the billing city of ‘New York’ matches the number of records inserted (i.e., 10).

Best Practices

As with future methods, there are a few things you want to keep in mind when using Batch Apex. To ensure fast execution of batch jobs, minimize Web service callout times and tune queries used in your batch Apex code. The longer the batch job executes, the more likely other queued jobs are delayed when many jobs are in the queue. Best practices include:

- Only use Batch Apex if you have more than one batch of records. If you don't have enough records to run more than one batch, you are probably better off using Queueable Apex.
- Tune any SOQL query to gather the records to execute as quickly as possible.
- Minimize the number of asynchronous requests created to minimize the chance of delays.
- Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger won’t add more batch jobs than the limit.

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Controlling Processes with Queueable Apex

Learning Objectives

After completing this unit, you'll know:

- When to use the Queueable interface.
- The differences between queueable and future methods.
- Queueable Apex syntax.
- Queueable method best practices.

Queueable Apex

Released in Winter '15, Queueable Apex is essentially a superset of future methods with some extra #awesomesauce. We took the simplicity of future methods and the power of Batch Apex and mixed them together to form Queueable Apex! It gives you a class structure that the platform serializes for you, a simplified interface without `start` and `finish` methods and even allows you to utilize more than just primitive arguments! It is called by a simple `System.enqueueJob()` method, which returns a job ID that you can monitor. It beats sliced bread hands down!

Queueable Apex allows you to submit jobs for asynchronous processing similar to future methods with the following additional benefits:

- Non-primitive types: Your Queueable class can contain member variables of non-primitive data types, such as sObjects or custom Apex types. Those objects can be accessed when the job executes.
- Monitoring: When you submit your job by invoking the `System.enqueueJob` method, the method returns the ID of the `AsyncApexJob` record. You can use this ID to identify your job and monitor its progress, either through the Salesforce user interface in the Apex Jobs page, or programmatically by querying your record from `AsyncApexJob`.
- Chaining jobs: You can chain one job to another job by starting a second job from a running job. Chaining jobs is useful if you need to do some sequential processing.

Queueable Versus Future

Because queueable methods are functionally equivalent to future methods, most of the time you'll probably want to use queueable instead of future methods. However, this doesn't necessarily mean you should go back and refactor all your future methods right now. If you were exceeding a governor limit in your future method, or if you think a future method requires a higher limit, you can possibly increase the limits for your future method with the Future Methods with Higher Limits pilot.

Another reason to use future methods instead of queueable is when your functionality is sometimes executed synchronously, and sometimes asynchronously. It's much easier to refactor a method in this manner than converting to a queueable class. This is handy when you discover that part of your existing code needs to be moved to async execution. You can simply create a similar future method that wraps your synchronous method like so:

```
@future
static void myFutureMethod(List<String> params)
{
    // call synchronous method
    mySyncMethod(params);
}
```

Queueable Syntax

To use Queueable Apex, simply implement the Queueable interface.

```
public class SomeClass implements Queueable {
    public void execute(QueueableContext context)
{
        // awesome code here
    }
}
```

Sample Code

A common scenario is to take some set of sObject records, execute some processing such as making a callout to an external REST endpoint or perform some calculations and then update them in the database asynchronously. Since `@future` methods are limited to primitive data types (or arrays or collections of primitives), queueable Apex is an ideal choice. The following code takes a collection of Account records, sets the parentId for each record, and then updates the records in the database.

```
public class UpdateParentAccount implements Queueable {

    private List<Account> accounts;
    private ID parent;

    public UpdateParentAccount(List<Account> records, ID id)
{
        this.accounts = records;
        this.parent = id;
    }

    public void execute(QueueableContext context) {
        for (Account account : accounts) {
          account.parentId = parent;
          // perform other processing or callout
        }
        update accounts;
    }

}
```

To add this class as a job on the queue, execute the following code:

```
// find all accounts in 'NY'
List<Account> accounts = [select id from account where billingstate =
'NY'];
// find a specific parent account for all records
Id parentId = [select id from account where name = 'ACME Corp'][0].Id;

// instantiate a new instance of the Queueable class
UpdateParentAccount updateJob = new UpdateParentAccount(accounts,
parentId);

// enqueue the job for processing
ID jobID = System.enqueueJob(updateJob);
```

After you submit your queueable class for execution, the job is added to the queue and will be processed when system resources become available.

You can use the new job ID to monitor progress, either through the Apex Jobs page or programmatically by querying `AsyncApexJob`:

```
SELECT Id, Status, NumberOfErrors FROM AsyncApexJob WHERE Id =
:jobID
```

Testing Queueable Apex

The following code sample shows how to test the execution of a queueable job in a test method. It looks very similar to Batch Apex testing. To ensure that the queueable process runs within the test method, the job is submitted to the queue between the `Test.startTest` and `Test.stopTest` block. The system executes all asynchronous processes started in a test method synchronously after the `Test.stopTest` statement. Next, the test method verifies the results of the queueable job by querying the account records that the job updated.

```
@isTest
public class UpdateParentAccountTest {

    @testSetup
    static void setup() {
        List<Account> accounts = new List<Account>();
        // add a parent account
        accounts.add(new Account(name='Parent'));
        // add 100 child accounts
        for (Integer i = 0; i < 100; i++) {
            accounts.add(new Account(
                name='Test Account'+i
            ));
        }
        insert accounts;
    }

    static testmethod void testQueueable() {
        // query for test data to pass to queueable class
        Id parentId = [select id from account where name = 'Parent'][0].Id;
        List<Account> accounts = [select id, name from account where name like 'Test
Account%'];
        // Create our Queueable instance
        UpdateParentAccount updater = new UpdateParentAccount(accounts, parentId);
        // startTest/stopTest block to force async processes to run
        Test.startTest();
        System.enqueueJob(updater);
        Test.stopTest();
        // Validate the job ran. Check if record have correct parentId now
        System.assertEquals(100, [select count() from account where parentId = :parentId]);
    }

}
```

Chaining Jobs

One of the best features of Queueable Apex is job chaining. If you ever need to run jobs sequentially, Queueable Apex could make your life much easier. To chain a job to another job, submit the second job from the `execute()` method of your queueable class. You can add only one job from an executing job, which means that only one child job can exist for each parent job. For example, if you have a second class called SecondJob that implements the Queueable interface, you can add this class to the queue in the `execute()` method as follows:

```
public class FirstJob implements Queueable {
    public void execute(QueueableContext context) {
        // Awesome processing logic here
        // Chain this job to next job by submitting the next
job
        System.enqueueJob(new SecondJob());
    }
}
```

Once again, testing has a slightly different pattern. You can't chain queueable jobs in an Apex test, doing so results in an error. To avoid nasty errors, you can check if Apex is running in test context by calling `Test.isRunningTest()` before chaining jobs.

Things to Remember

Queueable Apex is a great new tool but there are a few things to watch out for:

- The execution of a queued job counts once against the shared limit for asynchronous Apex method executions.
- You can add up to 50 jobs to the queue with `System.enqueueJob` in a single transaction.
- When chaining jobs, you can add only one job from an executing job with `System.enqueueJob`, which means that only one child job can exist for each parent queueable job. Starting multiple child jobs from the same queueable job is a no-no.
- No limit is enforced on the depth of chained jobs, which means that you can chain one job to another job and repeat this process with each new child job to link it to a new child job. However, for Developer Edition and Trial orgs, the maximum stack depth for chained jobs is 5, which means that you can chain jobs four times and the maximum number of jobs in the chain is 5, including the initial parent queueable job.

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Scheduling Jobs Using the Apex Scheduler

Learning Objectives

After completing this unit, you'll know:

- When to use scheduled Apex.
- How to monitor scheduled jobs.
- Scheduled Apex syntax.
- Scheduled method best practices.

Scheduled Apex

The Apex Scheduler lets you delay execution so that you can run Apex classes at a specified time. This is ideal for daily or weekly maintenance tasks using Batch Apex. To take advantage of the scheduler, write an Apex class that implements the Schedulable interface, and then schedule it for execution on a specific schedule.

Scheduled Apex Syntax

To invoke Apex classes to run at specific times, first implement the `Schedulable` interface for the class. Then, schedule an instance of the class to run at a specific time using the `System.schedule` method.

```
global class SomeClass implements Schedulable {
    global void execute(SchedulableContext ctx)
{
        // awesome code here
    }
}
```

The class implements the `Schedulable` interface and must implement the only method that this interface contains, which is the `execute` method.

The parameter of this method is a `SchedulableContext` object. After a class has been scheduled, a `CronTrigger` object is created that represents the scheduled job. It provides a `getTriggerId` method that returns the ID of a `CronTrigger` API object.

Sample Code

This class queries for open opportunities that should have closed by the current date, and creates a task on each one to remind the owner to update the opportunity.

```
global class RemindOpptyOwners implements Schedulable {

    global void execute(SchedulableContext ctx) {
        List<Opportunity> opptys = [SELECT Id, Name, OwnerId,
CloseDate
            FROM Opportunity
            WHERE IsClosed = False AND
            CloseDate < TODAY];
        // Create a task for each opportunity in the list
        TaskUtils.remindOwners(opptys);
    }

}
```

You can schedule your class to run either programmatically or from the Apex Scheduler UI.

Using the System.Schedule Method

After you implement a class with the Schedulable interface, use the `System.Schedule` method to execute it. The `System.Schedule` method uses the user's timezone for the basis of all schedules, but runs in system mode—all classes are executed, whether or not the user has permission to execute the class.

Note

Use extreme care if you're planning to schedule a class from a trigger. You must be able to guarantee that the trigger won't add more scheduled job classes than the limit. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

The `System.Schedule` method takes three arguments: a name for the job, a CRON expression used to represent the time and date the job is scheduled to run, and the name of the class.

```
RemindOpptyOwners reminder = new RemindOpptyOwners();
// Seconds Minutes Hours Day_of_month Month Day_of_week
optional_year
String sch = '20 30 8 10 2 ?';
String jobID = System.schedule('Remind Opp Owners', sch, reminder);
```

For more information on the CRON expression used for scheduling, see the "Using the System.Schedule Method" section in Apex Scheduler.

Scheduling a Job from the UI

You can also schedule a class using the user interface.

1. From Setup, enter `Apex` in the Quick Find box, then select **Apex Classes**.
2. Click **Schedule Apex**.
3. For the job name, enter something like `Daily Oppty Reminder`.
4. Click the lookup button next to Apex class and enter `*` for the search term to get a list of all classes that can be scheduled. In the search results, click the name of your scheduled class.
5. Select Weekly or Monthly for the frequency and set the frequency desired.
6. Select the start and end dates, and a preferred start time.
7. Click **Save**.

Testing Scheduled Apex

Just like with the other async methods we've covered so far, with Scheduled Apex you must also ensure that the scheduled job is finished before testing against the results. To do this, use `startTest` and `stopTest` again around the `System.schedule` method, to ensure processing finishes before continuing your test.

```
@isTest
private class RemindOppyOwnersTest {

    // Dummy CRON expression: midnight on March 15.
    // Because this is a test, job executes
    // immediately after Test.stopTest().
    public static String CRON_EXP = '0 0 0 15 3 ? 2022';

    static testmethod void testScheduledJob() {

        // Create some out of date Opportunity records
        List<Opportunity> opptys = new List<Opportunity>();
        Date closeDate = Date.today().addDays(-7);
        for (Integer i=0; i<10; i++) {
            Opportunity o = new Opportunity(
                Name = 'Opportunity ' + i,
                CloseDate = closeDate,
                StageName = 'Prospecting'
            );
            opptys.add(o);
        }
        insert opptys;

        // Get the IDs of the opportunities we just inserted
        Map<Id, Opportunity> opptyMap = new Map<Id, Opportunity>(opptys);
        List<Id> opptyIds = new List<Id>(opptyMap.keySet());

        Test.startTest();
        // Schedule the test job
        String jobId = System.schedule('ScheduledApexTest',
            CRON_EXP,
            new RemindOpptyOwners());
        // Verify the scheduled job has not run yet.
        List<Task> lt = [SELECT Id
            FROM Task
            WHERE WhatId IN :opptyIds];
        System.assertEquals(0, lt.size(), 'Tasks exist before job has
run');
        // Stopping the test will run the job synchronously
        Test.stopTest();

        // Now that the scheduled job has executed,
        // check that our tasks were created
        lt = [SELECT Id
            FROM Task
            WHERE WhatId IN :opptyIds];
        System.assertEquals(opptyIds.size(),
            lt.size(),
            'Tasks were not created');

    }
}
```

Things to Remember

Scheduled Apex has a number of items you need to be aware of (see Apex Scheduler in the Resources section for a complete list when you have time), but in general:

- You can only have 100 scheduled Apex jobs at one time and there are maximum number of scheduled Apex executions per a 24-hour period. See Execution Governors and Limits in the Resources section for details.
- Use extreme care if you're planning to schedule a class from a trigger. You must be able to guarantee that the trigger won't add more scheduled jobs than the limit.
- Synchronous Web service callouts are not supported from scheduled Apex. To be able to make callouts, make an asynchronous callout by placing the callout in a method annotated with `@future(callout=true)` and call this method from scheduled Apex. However, if your scheduled Apex executes a batch job, callouts are supported from the batch class.

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Monitoring Asynchronous Apex

Learning Objectives

After completing this unit, you'll know:

- How to monitor the different types of jobs.
- How to use the flex queue.

Monitoring Asynchronous Jobs

The great thing about async jobs is that they work silently in the background. The tough thing about async jobs is that they work silently in the background. Luckily there are a few ways to monitor what is going on with your jobs under the covers.

You can monitor the status of all jobs in the Salesforce user interface by going to **Setup** and clicking .

You can also monitor the status of Apex jobs in the Apex Flex Queue, and reorder them to control which jobs are processed first. You can get to this queue by going to **Setup** and clicking .

Monitoring Future Jobs

Future jobs show up on the Apex Jobs page like any other jobs. However, future jobs are not part of the flex queue at this time.

You can query `AsyncApexJob` to find your future job, but there's a caveat. Since kicking off a future job does not return an ID, you'll have to filter on some other field like `MethodName`, `JobType`, etc. to find your job. There are a few sample SOQL queries in this Stack Exchange post that may help.

Monitoring Queued Jobs with SOQL

To query information about your submitted job, perform a SOQL query on AsyncApexJob by filtering on the job ID that the `System.enqueueJob` method returns.

```
AsyncApexJob jobInfo = [SELECT Status,
NumberOfErrors
    FROM AsyncApexJob WHERE Id = :jobID];
```

Monitoring Queue Jobs with the Flex Queue

The Apex Flex queue enables you to submit up to 100 batch jobs for execution. Any jobs that are submitted for execution are in holding status and are placed in the Apex Flex queue. Up to 100 batch jobs can be in the holding status.

Jobs are processed first-in first-out—in the order in which they're submitted. You can look at the current queue order and shuffle the queue, so that you could move an important job to the front, or less important ones to the back.

When system resources become available, the system picks up the next job from the top of the Apex Flex queue and moves it to the batch job queue. The system can process up to five queued or active jobs simultaneously for each organization. The status of these moved jobs changes from Holding to Queued. Queued jobs get executed when the system is ready to process new jobs. Like other jobs, you can monitor queued jobs in the Apex Jobs page.

Monitoring Scheduled Jobs

After an Apex job has been scheduled, you can obtain more information about it by running a SOQL query on `CronTrigger`. The following sample queries the number of times the job has run, and the date and time when the job is scheduled to run again. It uses a jobID variable which is returned from the `System.schedule` method.

```
CronTrigger ct = [SELECT TimesTriggered, NextFireTime FROM CronTrigger WHERE Id =
:jobID];
```

If you're performing this query inside the execute method of your schedulable class, you can obtain the ID of the current job by calling `getTriggerId` on the `SchedulableContext` argument variable.

```
global class DoAwesomeStuff implements Schedulable {

    global void execute(SchedulableContext sc) {
        // some awesome code
        CronTrigger ct = [SELECT TimesTriggered, NextFireTime FROM CronTrigger WHERE Id =
:sc.getTriggerId()];
    }

}
```

You can also get the job's name and the job's type from the `CronJobDetail` record associated with the `CronTrigger` record. To do so, use the `CronJobDetail` relationship when performing a query on `CronTrigger`. This example retrieves the most recent `CronTrigger` record with the job name and type from CronJobDetail.

```
CronTrigger job = [SELECT Id, CronJobDetail.Id, CronJobDetail.Name, CronJobDetail.JobType FROM CronTrigger ORDER
BY CreatedDate DESC LIMIT 1];
```

Alternatively, you can query `CronJobDetail` directly to get the job's name and type. The following example gets the job's name and type for the `CronTrigger` record queried in the previous example. The corresponding `CronJobDetail` record ID is obtained by the `CronJobDetail.Id` expression on the `CronTrigger` record.

```
CronJobDetail ctd = [SELECT Id, Name, JobType FROM CronJobDetail WHERE Id =
:job.CronJobDetail.Id];
```

And lastly, to obtain the total count of all Apex scheduled jobs, excluding all other scheduled job types, perform the following query. Note the value '7' is specified for the job type, which corresponds to the scheduled Apex job type. See CronJobDetail in the Resources section below for a list of all types.

```
SELECT COUNT() FROM CronTrigger WHERE CronJobDetail.JobType =
'7'
```

Resources

Before You Start

Learning Objectives

After completing this unit, you'll be able to:

- Determine if you have the skills to complete this module.
- Configure My Domain for your Developer Edition org.
- Create a required custom object.

Before You Start This Module

We know. You're rarin' to get started. And far be it from us to dampen anyone's enthusiasm for Trailhead! But before you settle in to work through this module, we have a few things you should do. We suggest you do them *before* you plan to start the next units. These steps can take some time, while changes percolate through your org, and doing them might even send you off in another direction. So, it's worth working through this short unit first, before you commit to the full module.

The first thing we'll do is get My Domain enabled in the Developer Edition org you're using for your challenges. My Domain is required to develop with Lightning Components, and it can take a little time to activate—anywhere from 30 seconds to 30 minutes. We'd hate to eat up your Trailhead Time making you sit around and wait. We'll also create a necessary custom object, the Expense object, in your org.

Then, while we wait for My Domain, we can have a chat about whether you want to take this module on. Nobody likes to be excluded, but really, this module isn't for everyone. So we want to take a moment and talk about who this module *is* for, and skills you'll need to complete it.

We know this sounds like we're harshing on your excitement. We hate that as much as you do. But we want to be respectful of your time. If this isn't the right module for you, let's get you pointed at a different part of Trailhead, so your time with us is fun and challenging, without being frustrating.

Note

If you already have My Domain enabled in your DE org or you use a Trailhead Playground org, skip this section. You already have My Domain set up.

Add a Custom Domain to Your Org with My Domain

To use Lightning Components, your organization needs to have a custom domain configured using My Domain.

So what the heck is a custom domain, and why do you need to have one to use Lightning Components? First of all, a custom domain is a way to have your very own Salesforce server…sort of. It's a way for you to use Salesforce from your own, customized URL, rather than a generic Salesforce instance URL. That is, once you have a custom domain, you'll use Salesforce at https://*yourDomain*.my.salesforce.com/, which is reserved exclusively for your org's use. Let other folks continue to use and share https://na30.salesforce.com/. Your custom domain puts you on your own private Internet island.

Setting up a custom domain has a lot of benefits besides just getting you a cool URL. Among other things, a custom domain lets you:

- Highlight your business identity with your unique domain URL
- Brand your login screen and customize right-frame content
- Block or redirect page requests that don't use the new domain name
- Work in multiple Salesforce orgs at the same time
- Set custom login policy to determine how users are authenticated
- Let users log in using a social account, like Google and Facebook, from the login page
- Allow users to log in once to access external services

A custom domain also improves your organization's security, in ways too complicated to get into just now. And here we come to the reason it's required for Lightning Components. To provide world class security for apps, we're requiring that all users of Lightning Components use My Domain, just as we do for other advanced features, such as Salesforce Identity. If you want to use Lightning Components you must enable My Domain in your org.

Enable My Domain in Your Org

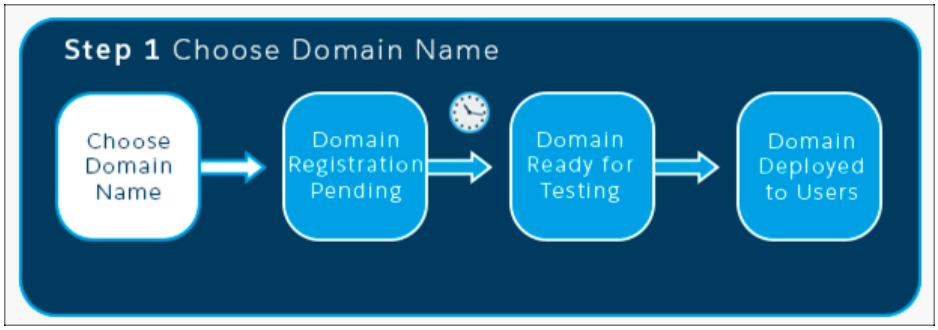
Before we get to the heart of creating Lightning components, let's use Salesforce My Domain to set up a subdomain. Is setting a My Domain a requirement? Yes, if you want to use Lightning components in Lightning tabs, Lightning Pages, or as standalone apps. Salesforce requires My Domain as a security measure to help prevent malicious attacks—just in case a security hole lies hidden deep within a third-party or custom component.

If you already have My Domain enabled in your DE org or use a Trailhead Playground org, skip this section and the next one. You already have My Domain set up.

If you don't have a subdomain yet, it's easy to set one up.

Every Salesforce org is set up within the salesforce.com domain with a URL like https://na30.salesforce.com. With My Domain, you define your own domain, or a subdomain, within the salesforce.com domain. Your new URL looks something like: https://*yourDomain*.my.salesforce.com.

Use the My Domain wizard to create a subdomain.



1. From Setup, enter `My Domain` in the Quick Find box, then select **My Domain**.
2. Enter the name for your subdomain after https:// and click **Check Availability**. Typically, a subdomain is your company name, but you can use any name as long as it's unique. If this name is already taken, choose another one.
3. Click **Register Domain**.

Salesforce updates its domain registries with your new subdomain. When it's done, you receive an email with a subject like, "Your Developer Edition

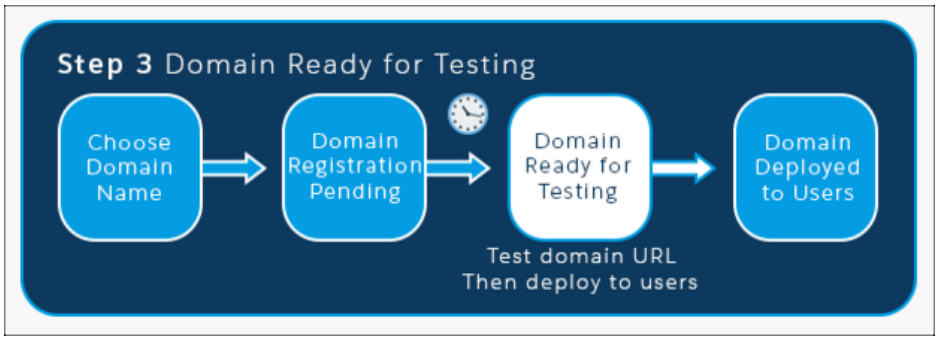
domain ready for testing.” It takes just a few minutes.

Important

Did you catch that last part? It can take a few minutes before your domain is available. You can’t move to the next step until you get the activation email.

Roll Out My Domain to Your Org

Did you get your activation email? From the email, click the link to get back to the My Domain wizard. It takes you to Step 3, where you test the links to your subdomain URLs before rolling out the subdomain to your org. Even though you don’t have users to deploy it to in your DE org, you must still roll out My Domain to make your custom Lightning components available in Lightning Pages, in the Lightning App Builder, and for standalone apps.



1. Click the link in the activation email to log in to your Salesforce subdomain. It takes you to your Salesforce org.
   Notice that the URL in the browser address bar shows your new subdomain name. Right now, you’re the only one who has this URL.

2. Click around your org to make sure that links point to your new domain. You probably haven’t created links in your DE org, so we can go on. (When creating a domain in a production org, this important step is easily overlooked.)
3. From the My Domain page, click **Deploy to Users**, and then click **OK**. Deploying a subdomain rolls out the new subdomain URL throughout your org. Now all your users see the subdomain URL in the browser address bar.
4. Step 4 of the wizard displays configuration options, which we can ignore for now.

Congratulations, you’ve set up My Domain! When setting up My Domain in a production org, you have a few more steps. Learn more by completing the My Domain unit of the User Authentication module. Now that you’ve protected—and branded—your org with a subdomain, let’s go on.

Define the Expense Custom Object

Many of the examples we’ll use in this module depend on a custom Expense object. You’ll get the most out of this module if you add these samples to your DE org and experiment with them for yourself. References to sObjects are validated on save and, if any object is undefined, the component is invalid. Lightning Components won’t let you save a component it knows is invalid. Let’s create the Expense object up front, so you don’t run into any problems compiling and saving code that depends on it.

While we assume that you know how to create a custom object already, here are brief instructions, and the specifics for the Expense object and its fields.

1. Go to the Object Manager.

   From Setup, enter “`object`” in the Quick Find box, and then select **Object Manager** under Create.

2. Create the custom object.

   Select .

3. Define the Expense object.
   Enter the following values for the object’s definition.

Field	Value
Label	Expense

Field	Value
Plural Label	Expenses
Starts with vowel sound	checked
API Name	Expense__c

Accept the defaults for the rest of the object definition.

4. Add custom fields to the Expense object.
   Scroll to the Fields & Relationships section of the object details page. For each of the following fields, click **New** and define the field with the following details.

Field Label	API Name	Field Type
Amount	Amount__c	Number(16,2)
Client	Client__c	Text(50)
Date	Date__c	Date
Reimbursed	Reimbursed__c	Checkbox

Skills You Need to Complete This Module

One of the great things about Salesforce is how much you can customize it using the app. Custom objects and fields, formulas, flows, reports, approvals, and even the user interface itself—you can do all of this and more from Setup, without writing a line of code, and make your users very happy.

But there are some features of Salesforce that require code, and Lightning Components is one of them. There are no two ways about this: to be successful with Lightning Components, you need to be able to read and write code. We'll be looking at a lot of code throughout this module, and you'll have to write a fair bit yourself to pass the challenges.

Specifically, we think that:

- You should be comfortable reading and writing JavaScript. Although Lightning Components also uses HTML-style markup, doing anything beyond "hello world" **requires** JavaScript.

  There's an enormous number of resources, free and otherwise, for learning JavaScript. We'd recommend getting started at JavaScript the Right Way. But if your friend or colleague has a recommendation, or a book they can loan you, go for it!

- It would be great if you know Apex. Reading and writing data from Salesforce usually uses Apex, and that's what we'll learn in this module. You can pass the challenges in this module without being an Apex guru, but when you go to write real apps, you'll be writing plenty of Apex.

  The Apex Basics & Database module is a great way to get started with Apex, and will give you everything you need for this module.

If this doesn't describe you, we don't want to turn you away. It's not impossible to complete this module without the preceding skills. But we do think it will be frustrating. And while you might earn the badge—which is definitely cool!—you won't really be ready to use Lightning Components to write apps.

We want Trailhead to be fun, and we want it to help people use Salesforce more completely and confidently. Experienced programmers will get a lot out of this module. But grinding through it without the requisite programming background might not be the best use of your Trailhead Time.

OK. With that out of the way, let's dive in and start learning!

Resources

Remember, this module is meant for Lightning Experience. When you launch your hands-on org, switch to Lightning Experience to complete this challenge.

Get Started with Lightning Components

Learning Objectives

After completing this unit, you'll be able to:

- Describe what the Lightning Components framework is, and what it's used for.
- List four key differences between Lightning Components and other web app frameworks.

- List at least five different ways you can use Lightning Components to customize Salesforce.

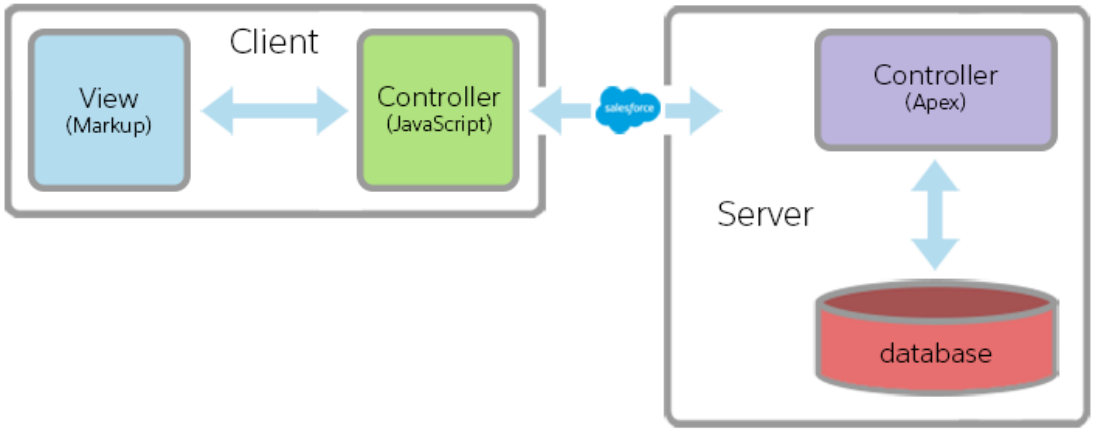
Getting Started with Lightning Components

Yes! You're still with us! We're really excited to welcome you to the Lightning Components party—and it *is* a party. When we say that Lightning Components is the most exciting and powerful app development technology we've built in years, it's a bold statement. And we know it's true, because we built Salesforce1 and Lightning Experience with it. We think that once you get to know Lightning Components, you'll be as excited as we are about using it.

What Is the Lightning Components Framework?

Lightning Components is a UI framework for developing web apps for mobile and desktop devices. It's a modern framework for building single-page applications with dynamic, responsive user interfaces for Force.com apps. It uses JavaScript on the client side and Apex on the server side.

That's…a lot of buzzwords. Let's see if we can't try that again, with words that normal people use.



"Lightning Components is a framework for developing web apps." That seems understandable. An app framework is a collection of code and services that make it easier for you to create your own custom apps, without having to write all the code yourself. There are lots of different web app frameworks out there, like Ruby on Rails, Grails, AngularJS, Django, CakePHP, and on and on. We've even got one of our own, Visualforce, that customers know and love. Lightning Components is new, and we think it's pretty special. We'll talk about why it's special more as we go, and hopefully by the end of this module, you'll agree!

"Web apps for mobile and desktop devices." Again, that seems pretty easy to grasp. But…did you notice the order there? Lightning Components was born out of and used to build the Salesforce1 platform for mobile apps. Mobile is baked into the core of Lightning Components, and it makes developing apps that work on both mobile and desktop devices far simpler than many other frameworks.

"It's a modern framework for building single-page applications." OK, now we're getting a little buzzed. "Modern" is just marketing, right? And what are "single-page applications"?

We don't think modern is "just" marketing. In the Develop for Lightning Experience module we talk at length about how web apps have evolved from simple, page-by-page oriented experiences to highly responsive apps that have all the same behavior and interactivity of native apps, on desktops and especially on mobile devices. To achieve these interactive user experiences, modern web apps are built as a tightly bound collection of code that loads from a single URL, and then runs continuously as you use it. These single-page apps are built much like native apps are, with the plumbing being handled by a framework. A framework like Lightning Components.

"Dynamic, responsive user interfaces for Force.com apps" is just applying the preceding ideas to apps you build on top of Salesforce. And finally, "it uses JavaScript on the client side and Apex on the server side" is pretty self-explanatory by itself—even if it leaves out some specifics about what goes where. We'll get to that, soon!

An Example Lightning Component

OK, that's a lot of talk talk talk, and not a lot of code. Let's take a look at a real Lightning component, and see what all that talk is about. First, here's what the component looks like when rendered on screen:

It might not look like much, but there's a fair bit going on. Here's the code for it; this is from a component we'll dig into in detail later.

```
<aura:component>

    <aura:attribute name="expense" type="Expense__c"/>
    <aura:registerEvent name="updateExpense" type="c:expensesItemUpdate"/>

    <div class="slds-card">
        <!-- Color the item blue if the expense is reimbursed -->
        <div class="{!v.expense.Reimbursed__c == true ?
            'slds-theme--success' : 'slds-theme--warning'}">

            <header class="slds-card__header slds-grid grid--flex-spread">
                <a aura:id="expense" href="{!'/' + v.expense.Id}">
                    <h3>{!v.expense.Name}</h3>
                </a>
            </header>

            <section class="slds-card__body">
                <div class="slds-tile slds-hint-parent">
                    <p class="slds-tile__title slds-truncate">Amount:
                        <ui:outputCurrency value="
{!v.expense.Amount__c}"/>
                    </p>
                    <p class="slds-truncate">Client:
                        <ui:outputText value="{!v.expense.Client__c}"/>
                    </p>
                    <p class="slds-truncate">Date:
                        <ui:outputDate value="{!v.expense.Date__c}"/>
                    </p>
                    <p class="slds-truncate">Reimbursed?
                        <ui:inputCheckbox value="
{!v.expense.Reimbursed__c}"
                                click="{!c.clickReimbursed}"/>
                    </p>
                </div>
            </section>
        </div>
    </div>

</aura:component>
```

Even before you know anything about Lightning Components, you can still notice a few things about this sample. First of all, it's XML markup, and mixes both static HTML tags with custom Lightning Components tags, such as the tag that leads off the sample. If you've worked with Visualforce, the format of that tag is familiar: *`namespace`*`:`*`tagName`*. As you'll see later, built-in components can come from a variety of different namespaces, such as `aura:` (as here), or `force:`, `lightning:`, or `ui:`.

Speaking of `ui:`, you might have noticed that there are input and output components, like and . Again, this is a pattern familiar to Visualforce developers. If you're not one of those, hopefully it's pretty obvious that you use the input components to collect user input, and the output components to display read-only values.

We'll get to the rest of the components in later units. For now, one last thing to notice is the use of static HTML with a number of CSS class names that start with "slds". We're going to use the Salesforce Lightning Design System, or SLDS, to style our components, and while we won't explain SLDS in detail in this module, we want you to see examples of it in action.

OK, cool, Lightning Components markup is XML. But didn't we say something about JavaScript earlier? Notice the `click="{!c.clickReimbursed}"` attribute on the checkbox? That means "when this checkbox is clicked, call the controller's `clickReimbursed` function." Let's look at the code it's attached to.

```
({
    clickReimbursed: function(component, event, helper) {
        var expense = component.get("v.expense");
        var updateEvent =
component.getEvent("updateExpense");
        updateEvent.setParams({ "expense": expense });
        updateEvent.fire();
    }
})
```

This is the component's client-side controller, written in JavaScript. The **`clickReimbursed`** function in the component's controller corresponds to the `click="{!c.`**`clickReimbursed`**`}"` attribute on the checkbox in the component's markup.

In Lightning Components, a component is a bundle of code. It can include markup like the earlier sample, in the ".cmp resource," and it can also include JavaScript code, in a number of associated resources. Related resources are "auto-wired" to each other, and together they make up the component bundle.

We'll get to the details in the next unit, but for now, you've seen the two most important types of Lightning Components code.

What About Visualforce?

The question we get from customers, over and over, is this: "Which should I use, Lightning Components or Visualforce?" The short answer is: yes!

Visualforce and Lightning Components each have their strengths. These are discussed in the Develop for Lightning Experience module, where you can find the long answer about appropriate uses for each. Here, let's go for medium.

First, know this: Visualforce isn't going away. Your Visualforce code will run on Salesforce for a long time to come. You don't *need* to convert existing Visualforce apps, and you don't *need* to stop creating apps with Visualforce.

However, you might *want* to, at least in some cases. For example, Visualforce was created before mobile apps on phones became a thing. While you *can* develop mobile apps with Visualforce, none of the built-in components are mobile-savvy. Which means you write more code. Lightning Components, on the other hand, is specifically optimized to perform well on mobile devices.

Again, we cover a lot of specifics in the Develop for Lightning Experience module. If you still have questions about Visualforce and Lightning Components, that's a good place to go next.

What About AngularJS, React, and Those Other JavaScript Frameworks?

Another question that comes up frequently is: "How does Lightning Components compare to *MyFavoriteFramework*?" where that favorite framework is another modern JavaScript web app framework such as AngularJS, React, or Ember.

These are all fine frameworks! Many people know them, and there are a lot of resources for learning them. You might be surprised to learn that we think these frameworks are a great way to build Force.com apps!

We recommend using them with Visualforce, using what we call a container page, and packaging your chosen framework and app code into static resources. Using an empty container page has Visualforce get out of your way, and lets you use the full capabilities of your chosen framework.

While it's possible to use third-party JavaScript frameworks with Lightning Components, it's a bit cumbersome. Lightning Components doesn't have the notion of an empty page, and has some specific opinions about how, for example, data access is performed, and some rather specific security requirements.

And frankly, the features of Lightning Components and most modern frameworks overlap quite a bit. While the *style* or *specifics* might be different, the *features* provided are conceptually similar enough that you're effectively running duplicate code. That's neither efficient nor easy to work with.

Another thing to consider: general-purpose frameworks such as AngularJS are designed to be agnostic about the platform they run on top of, in particular data services. Lightning Components, on the other hand, is designed to connect natively with services provided by Salesforce and the Force.com platform. Which do you think is going to help you build apps faster?

Note

We're only talking about **application frameworks** here. If you have a favorite JavaScript charting or mapping library, or other special-purpose toolkits then—subject to certain security requirements—modern JavaScript libraries usually work fine.

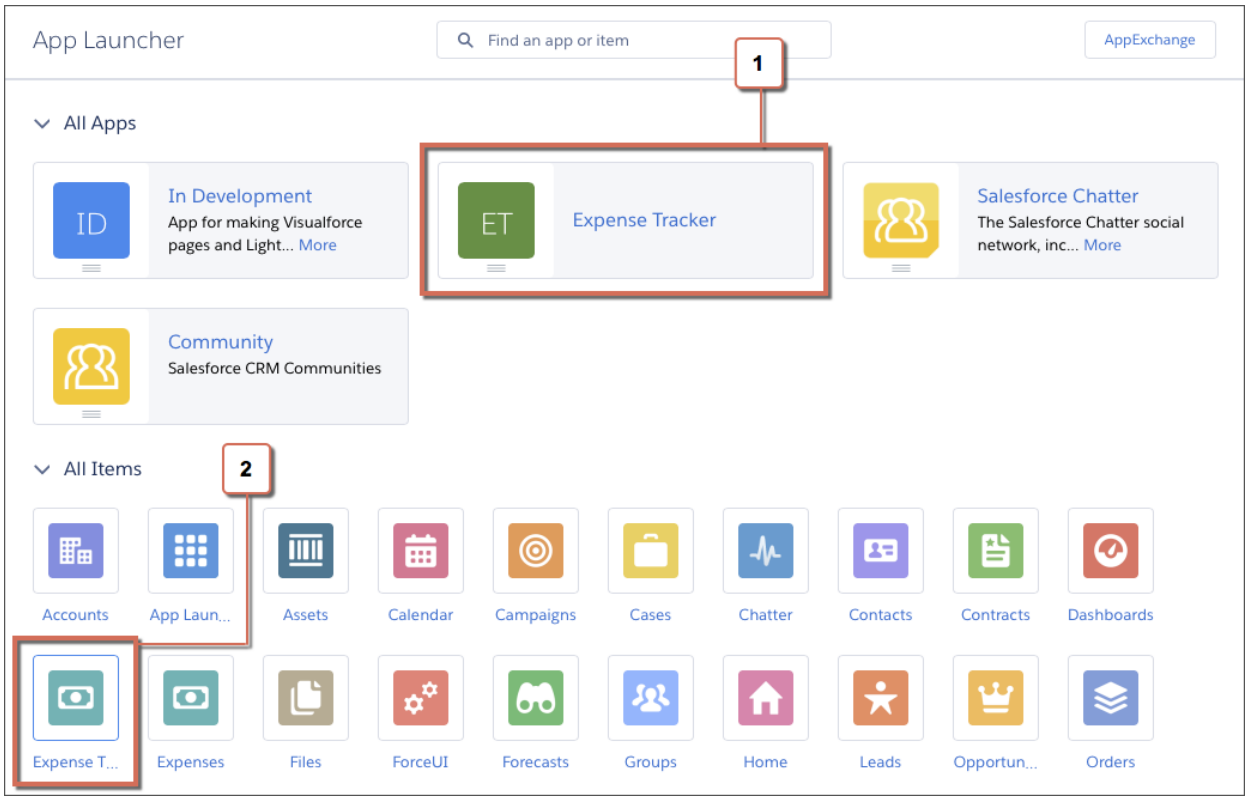
OK, enough words words words! Let's take a quick-and-graphical tour of the many places you can deploy Lightning Components apps. And then let's dive into the fun stuff: the code.

Where You Can Use Lightning Components

You can use Lightning Components to customize your Salesforce org in a number of different ways. But that's not all! You can use Lightning Components to create stand-alone apps that are hosted on Salesforce. And you can even create apps that are hosted on other platforms, including embedding them into apps from those platforms.

Add Apps to the Lightning Experience App Launcher

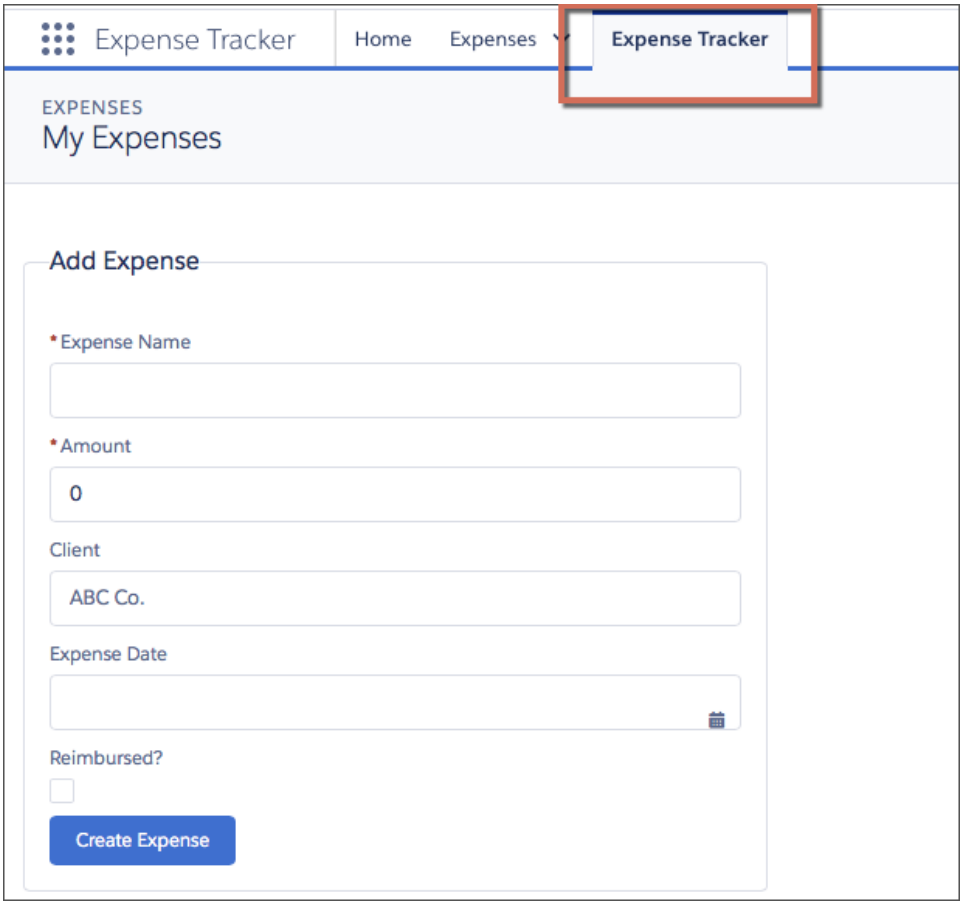
Your Lightning Components apps and custom tabs are available from the App Launcher, which you reach by clicking in the header.



Click a custom app (1) to activate it. Items in the app display in the navigation bar, including any Lightning components tabs you've added to the app. Note that you need to add your components to tabs for them to be accessible in the App Launcher. Lightning components tabs that aren't in apps can be found in All Items (2).

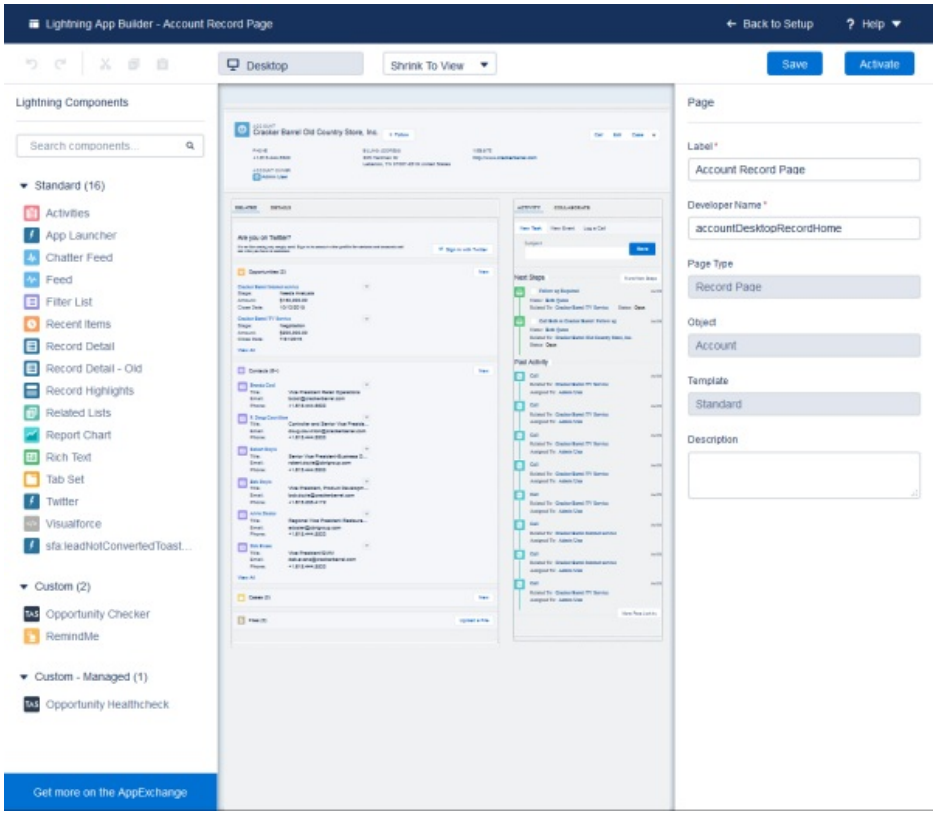
Add Apps to Lightning Experience and Salesforce1 Navigation

As described in the preceding example, you can add Lightning components tabs to an app and they display as items in the app's navigation bar.



Create Drag-and-Drop Components for Lightning App Builder and Community Builder

Build custom user interfaces using your own Lightning components, or those you install from AppExchange, for desktop and mobile devices.

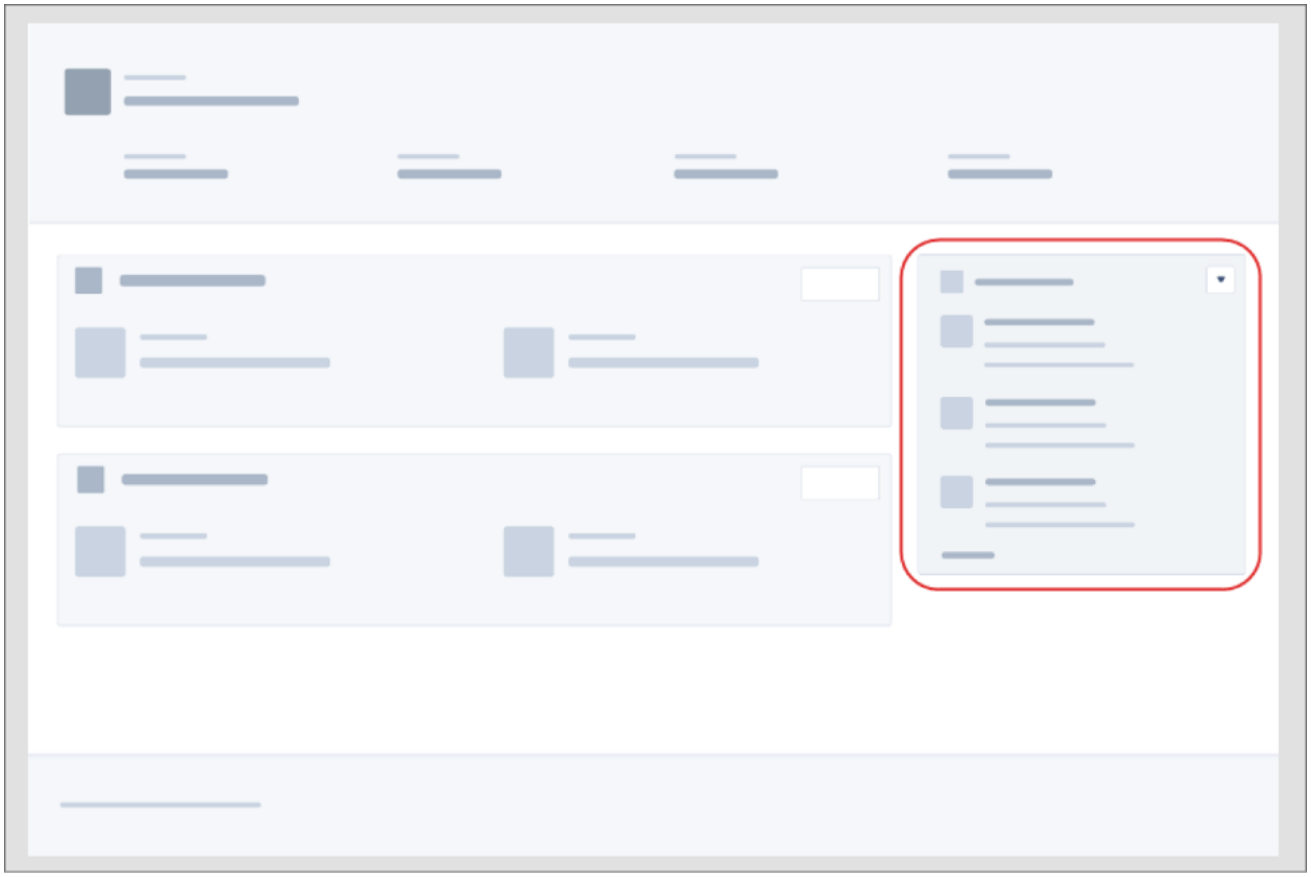


Add Lightning Components to Lightning Pages

A Lightning Page is a custom layout that lets you design pages for use in the Salesforce1 mobile app or in Lightning Experience. You can use a Lightning Page to create an app home page and add your favorite Lightning component, such as the Expenses app we'll be creating in this module, to it.

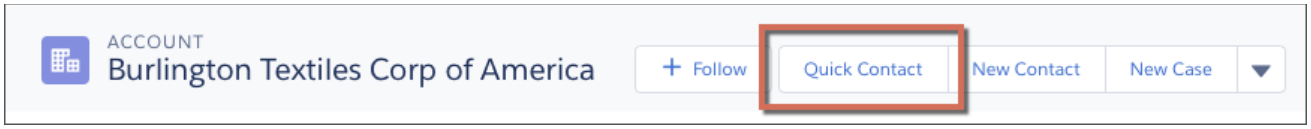
Add Lightning Components to Lightning Experience Record Pages

Just as the title suggests, you can customize Lightning Experience record pages by adding a Lightning Component.



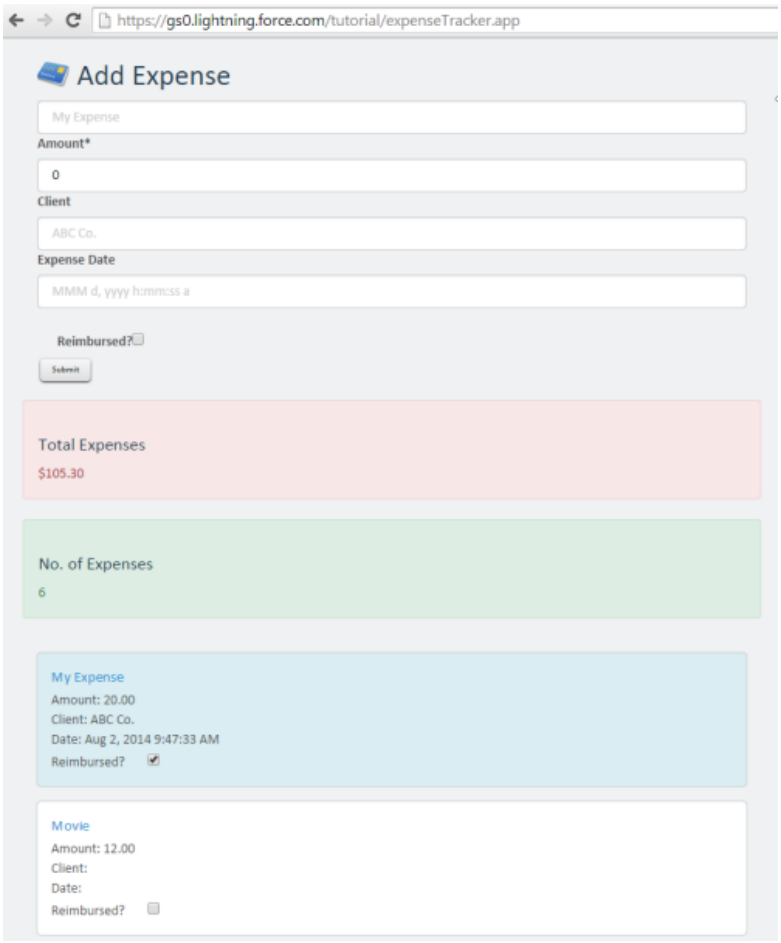
Launch a Lightning Component as a Quick Action

Create actions using a Lightning component, and then add the action to an object's page layout to make it instantly accessible from a record page.



Create Stand-Alone Apps

A standalone app comprises components that use your Salesforce data and can be used independently from the standard Salesforce environment.

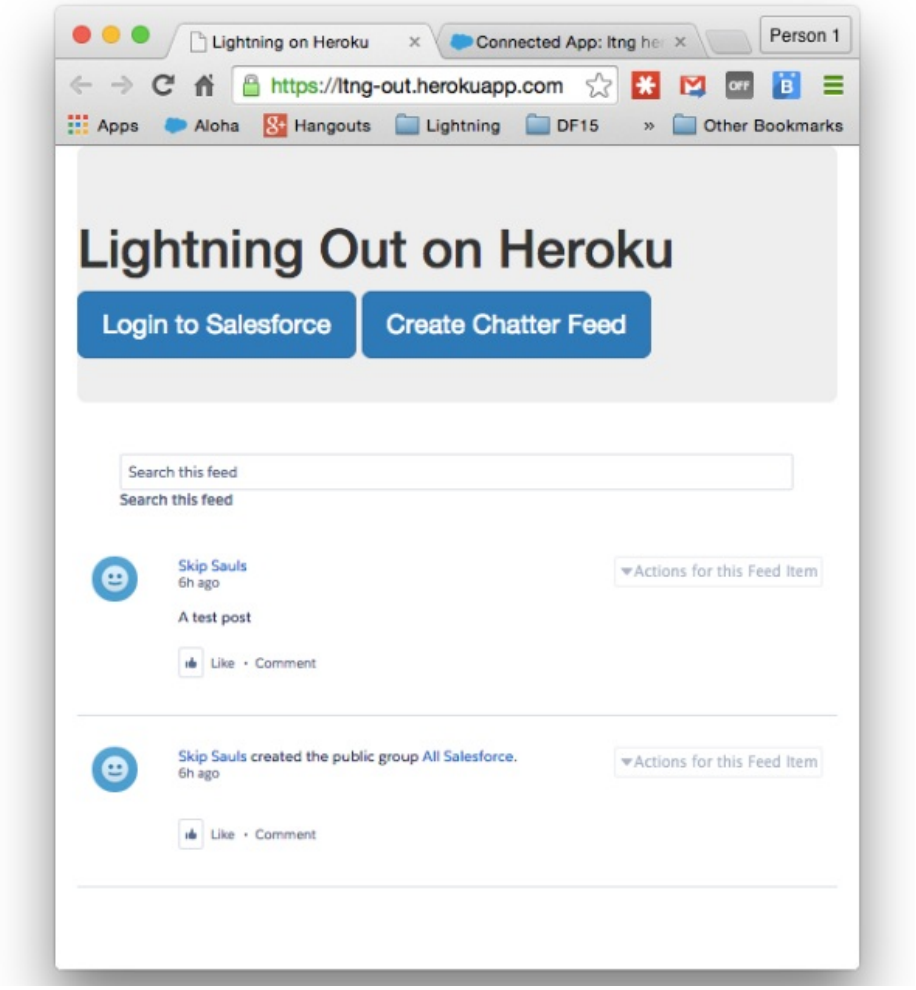


Run Lightning Components Apps Inside Visualforce Pages

Add Lightning components to your Visualforce pages to combine features you've built using both solutions. Implement new functionality using Lightning components and then use it with existing Visualforce pages.

Run Lightning Components Apps on Other Platforms with Lightning Out

Lightning Out is a feature that extends Lightning apps. It acts as a bridge to surface Lightning components in any remote web container. This means you can use your Lightning components inside of an external site (for example, Sharepoint or SAP), in a hybrid app built with the Mobile SDK, or even elsewhere in the App Cloud like on Heroku.



Resources

Create and Edit Lightning Components

Learning Objectives

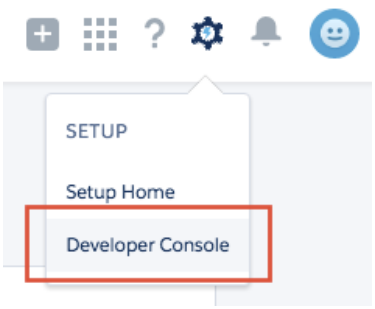
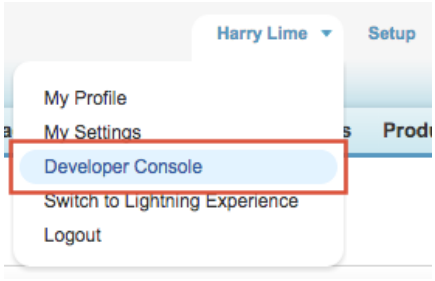
After completing this unit, you'll be able to:

- Create and edit Lightning component bundle resources in the Developer Console.
- Create a "harness" application for testing components in development.
- Perform the edit and reload cycle for previewing components in development.
- List the different resources that make up a Lightning component bundle.

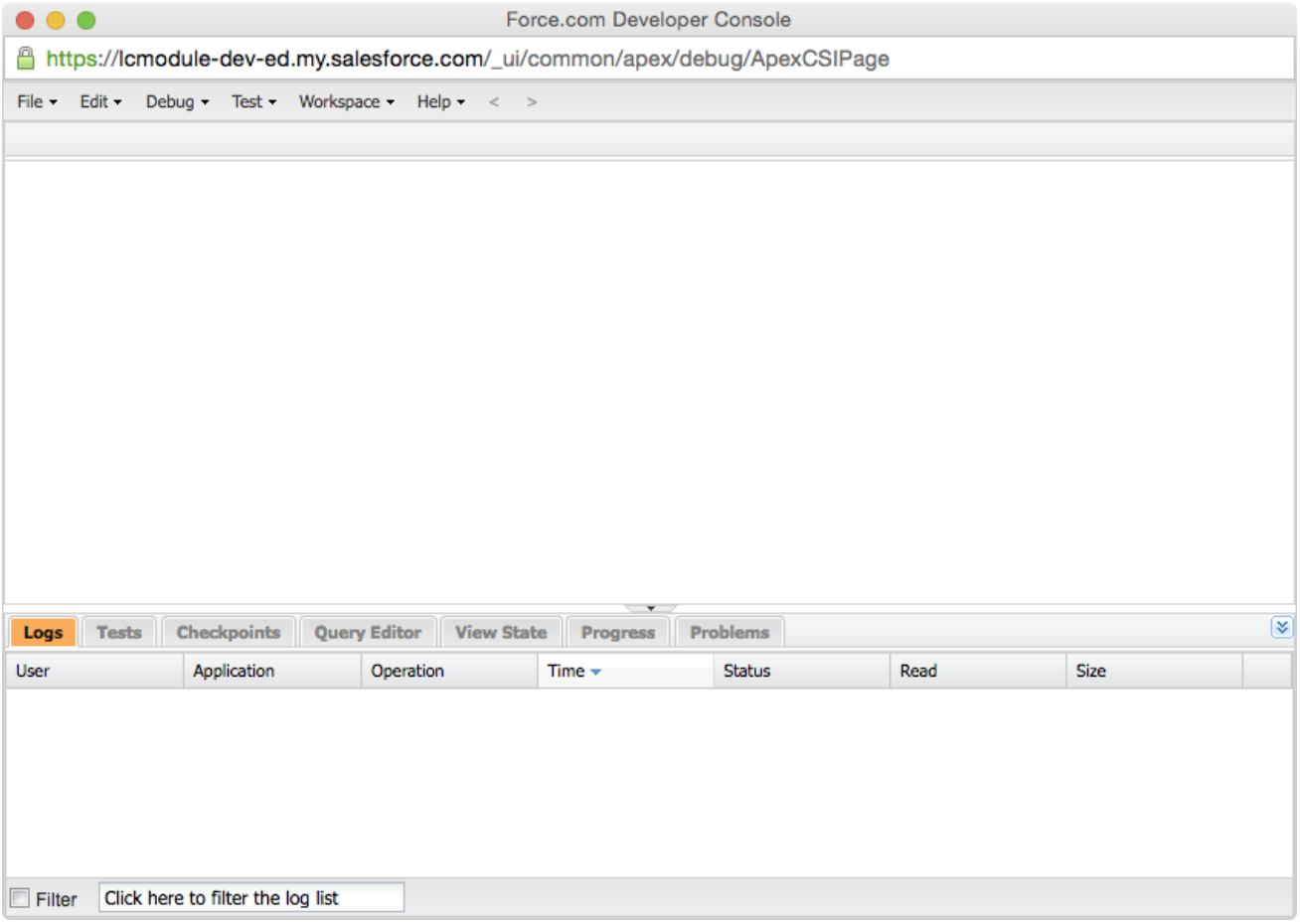
Creating and Editing Lightning Components

OK! Time to write some code! #finally!

The first step to writing Lightning Components code is, uh, getting set up to write code. Fortunately, this is really easy. In your DE org, open the Developer Console under *Your Name* or the quick access menu ( ).

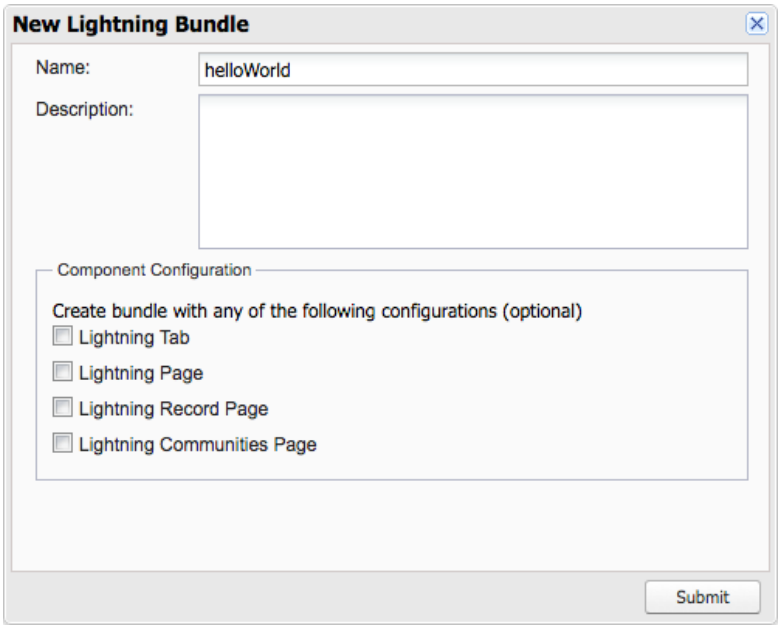
Lightning Experience	Salesforce Classic
	

Boom, you're ready to write Lightning Components code!



Create Lightning Components in the Developer Console

So, let's write something. Select to create a new Lightning component. In the **New Lightning Bundle** panel, enter `helloWorld` for the component name, and click **Submit**.

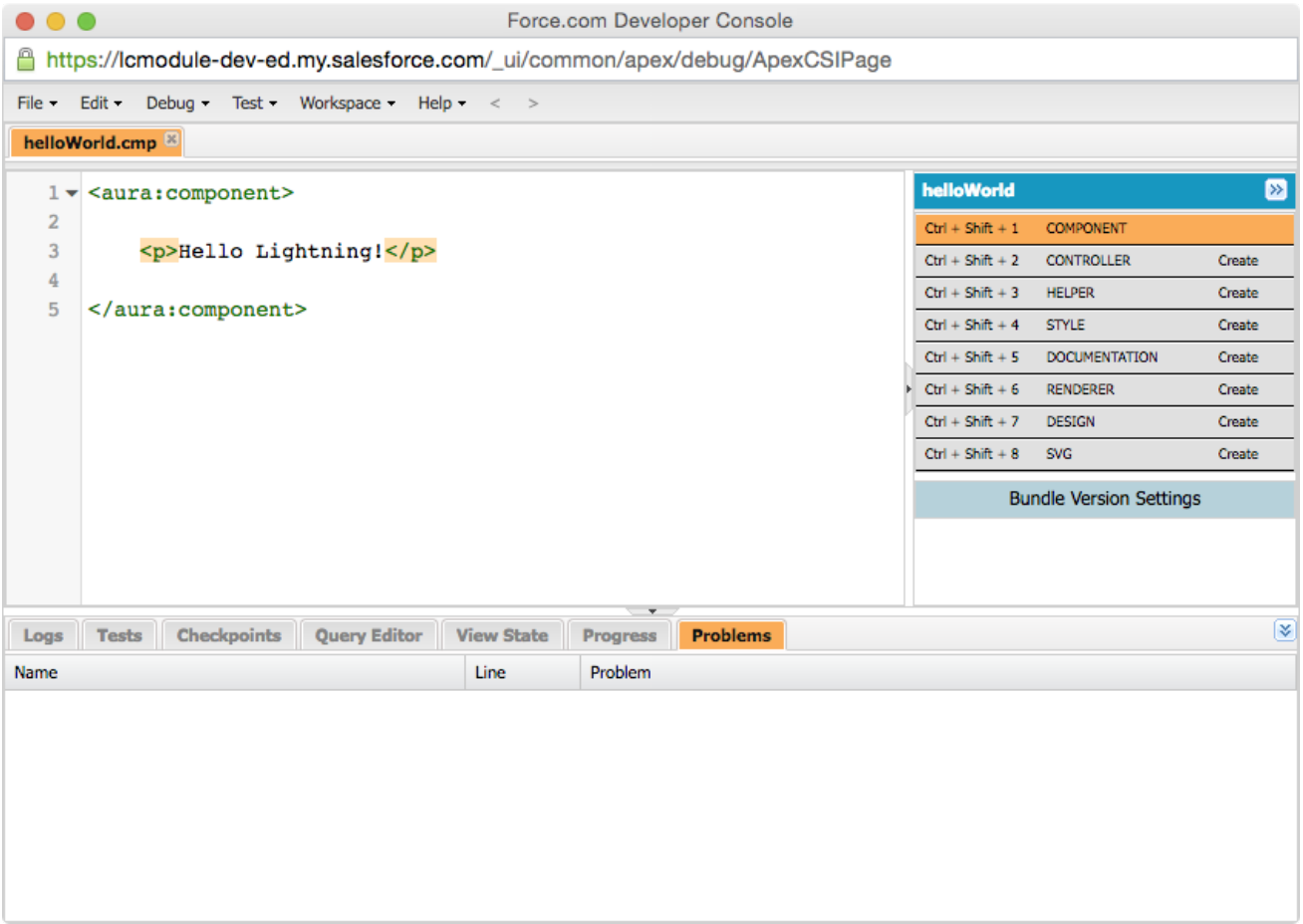


This creates a new `helloWorld` component bundle, with two open tabs. Close the `helloWorld` tab, and keep the `helloWorld.cmp` tab open.

`helloWorld.cmp` contains the opening and closing tags for a Lightning component, . Between them, add the following markup, and save:

```
<p>Hello Lightning!
</p>
```

Your component markup should look like the following.



Woohoo, your first Lightning component! Now…how do we see what it looks like?

The short answer is, it's tricky. You can't run your component directly and see how it behaves. Instead, your component needs to run inside a container app, which we'll call a container for short. Examples of containers would be the Lightning Experience or Salesforce1 apps, or an app you build with Lightning App Builder—basically, any of the things you saw at the end of the prior unit. You add your component to one of these containers, and then access it within that container.

We'll talk more about containers later, and in other Lightning Components modules. For now, let's just make a simple one of our own.

Select to create a new Lightning app. In the **New Lightning Bundle** panel, enter "`harnessApp`" for the app name, and click **Submit**.

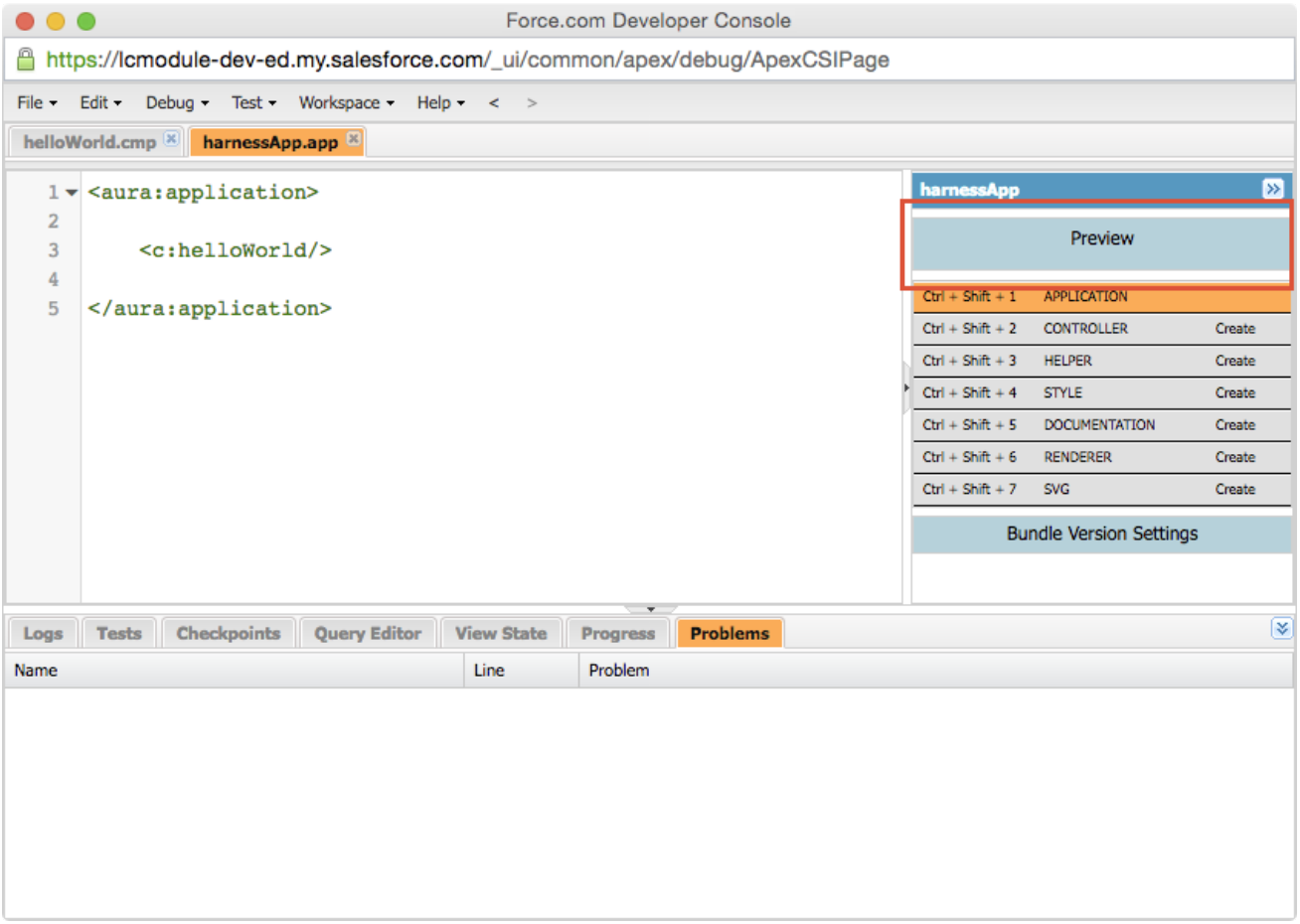
This creates a new `harnessApp` bundle, with two open tabs. Close the `harnessApp` tab, and keep the `harnessApp.app` tab open.

`harnessApp.app` contains the opening and closing tags for a Lightning app, . Between them, add the following markup, and save:

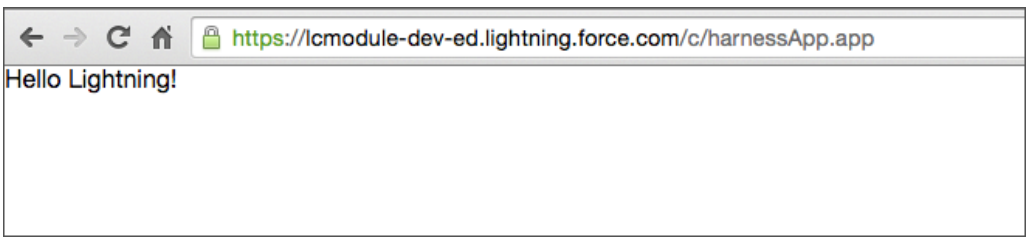
```
<c:helloWorld/>
```

This adds the `helloWorld` component we created earlier to the `harnessApp` app.

Before we explain this deceptively simple app, click back and forth between the `harnessApp.app` and `helloWorld.cmp` tabs in the Developer Console. Besides the markup, what do you notice that's different?



Got it in one: the **Preview** button. Apps have one, components don't. Click it now, and another browser window should open and show you your app.



Now we're cooking with…well, OK, it's just "hello world." But there are some interesting things to notice here, in spite of the markup being trivial.

Let's start with the Developer Console. If you've used it to write Visualforce or Apex, you have surely noticed the extra controls that appear in a palette on

the right side of the editing window of any Lightning bundle. Each of the different buttons with a **Create** label represents a different *resource* you can add to the bundle. We'll talk about resources and bundles in the next section. For now, just know that the Developer Console gives you an easy way to create and switch between them.

Indeed, the Developer Console has a number of features for working with Lightning Components. There's also , which lets you open a bunch of Lighting resources all at once. Useful!

The Developer Console is a great place to write Lightning code, and we'll work with it for the rest of this module. But, because Lightning resources are accessible via the Force.com Tooling API, there are other ways to create and edit them. The Force.com IDE is a good one, and there are excellent third-party tools such as Sublime Lightning and Mavens Mate. Don't feel like you're limited to just the Developer Console!

One last thing before we turn to the code. The URL for our "preview" is actually the permanent home of our app (once it's made available to our users). The format of the URL is the following: https://.lightning.force.com//.app.

represents the name of your app bundle, in this case, `harnessApp`. In your Trailhead DE orgs, you shouldn't have a namespace configured, so you should see a "c" in that part of the URL. "c" represents the default namespace…and it will come back to haunt us later. The rest of the URL format should be self-explanatory.

OK, on to the code!

What Is a Component?

It's not often that hello world triggers existential questions, but here we are. Let's talk about what a component is in the context of our `helloWorld` example. As a practical matter, a component is a bundle that includes a definition resource, written in markup, and may include additional, optional resources like a controller, stylesheet, and so on. A resource is sort of like a file, but stored in Salesforce rather than on a file system.

Our `helloWorld.cmp` component definition resource is easy to understand.

```
<aura:component>
    <p>Hello Lightning!
</p>
</aura:component>
```

There are the opening and closing tags, with some static HTML in between. It would be hard to be more simple, and you might be tempted to think of it as a "page." Don't. We'll come back to this in a bit.

We've mentioned component bundles already, but what are they really? A *bundle* is sort of like a folder. It groups the related resources for a single component. Resources in a bundle are *auto-wired* together via a naming scheme for each resource type. Auto-wiring just means that a component definition can reference its controller, helper, etc., and those resources can reference the component definition. They are hooked up to each other (mostly) automatically.

Let's see how this works. With `helloWorld.cmp` active, click the **STYLE** button in the component palette on the right. This opens a new tab for the style resource that was added to the `helloWorld` bundle. It starts with a single, empty selector, `.THIS`. To see how this works, add a simple style to the stylesheet, so that it looks like the following.

```
.THIS {
}

p.THIS {
    font-size:
24px;
}
```

Then reload your preview window for `harnessApp.app`. Voilà, larger text! But, how does `.THIS` work? It's the magic of auto-wiring! At runtime `.THIS` is replaced with a style scoping string named for your component. It limits style rules to *only* this component, so that you can create styles that are specific to the component, without worrying about how those styles might affect other components.

So now our `helloWorld` bundle has two resources, the component definition, `helloWorld.cmp`, and the stylesheet, `helloWorld.css`. You can think of it like a folder, or an outline:

- `helloWorld` — the component bundle
    - `helloWorld.cmp` — the component's definition
    - `helloWorld.css` — the component's styles

As you can see in the Developer Console, there are a number of other resource types you can add to a component bundle. Go ahead and click the **CONTROLLER** and **HELPER** items to add those resources to the bundle. Now your bundle looks something like this, and you can start to see the naming system.

- `helloWorld` — the component bundle
  - `helloWorld.cmp` — the component's definition
  - `helloWorldController.js` — the component's controller, or main JavaScript file
  - `helloWorldHelper.js` — the component's helper, or secondary JavaScript file
  - `helloWorld.css` — the component's styles

In this module, we'll work with only these four resource types. We'll talk a *lot* more about the controller and helper resources when we actually start writing code for them. For now, you can just leave the default implementations. After all, this is just hello world!

What Is an App?

Now that we know what a component is, it's actually easy to explain what an app is—an app is just a special kind of component! For the purposes of this module, you can think of an app as being different from a component in only two meaningful ways:

- An app uses tags instead of tags.
- Only an app has a **Preview** button in the Developer Console.

That's it!

What Are Apps For?

As simple as that sounds, there are a few practical details in how you can *use* an app vs. a component. The main items are the following.

- When writing markup, you can add a component to an app, but you **can't** add an app to another app, or an app to a component.
- An app has a standalone URL that you can access while testing, and which you can publish to your users. We often refer to these standalone apps as "my.app."
- You can't add apps to Lightning Experience or Salesforce1—you can only add components. After the last unit this might sound weird; what exactly do you add to the App Launcher, if not an app? What you add to App Launcher is a *Salesforce* app, which wraps up a Lightning component, something defined in a . A Lightning Components app—that is, something defined in a —can't be used to create Salesforce apps. A bit weird, but there it is.

So, what's an app good for? Why would you ever use one? We answered that question earlier. You publish functionality built with Lightning Components in *containers*. Lightning Components apps are one kind of container for our Lightning components.

Once again being practical, this usually means that you build all of your "app" functionality inside a top-level component. Then at the end, you stick that one component in a container—maybe a Lightning Components app, maybe Salesforce1, maybe something else. If you use a my.app, the container can set up services for your main component, but otherwise it's just there to host the component.

Let's take another look at the app we created. Here again is the `harnessApp.app` definition resource:

```
<aura:application>

<c:helloWorld/>

</aura:application>
```

No matter how much functionality we decide we're going to add to our helloWorld "app", it's all going to go inside the `helloWorld` component. It could have a Quip-style editor embedded in it for revising the hello message, but our `harnessApp.app` definition is going to remain pretty much this simple.

From here on, we'll assume that you're using an actual Lighting Application bundle as just a container, or harness, for components you create. Feel free to keep using `harnessApp.app`! But, when we talk about creating apps, we really mean building functionality inside a component bundle, not an application bundle, because that's how you build "apps" in the real world.

Components Containing Components, Containing…Components!

The `harnessApp.app` definition is also interesting because instead of static HTML we have our `helloWorld` component. We say that `harnessApp` contains the `helloWorld` component. Let's dive into this a little bit, and make `helloWorld` a little more complex.

In the Developer Console, create a new Lightning component named `helloHeading`. For its markup, paste in the following code.

```
<aura:component>
    <h1>W E L C O M
E</h1>
</aura:component>
```

Now, click back to helloWorld.cmp, and add to it, above the “Hello Lightning” line. Your `helloWorld` component definition should now look like this:

```
<aura:component>

    <c:helloHeading/>

    <p>Hello Lightning!
</p>

</aura:component>
```

Reload the app to see the change. Your component structure, in terms of what contains what, now looks like this:

- `harnessApp.app`
    - `helloWorld.cmp`
        - `helloHeading.cmp`
        - (static HTML)

We say that `helloHeading` is a child component of `helloWorld`, or that `helloHeading` is nested inside `helloWorld`, or…. There are any number of different ways to say that `helloWorld` contains `helloHeading`. What’s more, you can keep nesting components inside other components down to pretty much any level you’d care to. It starts being too hard to keep straight in your head well before you run into a limitation of Lightning Components!

This process of putting components inside each other is fundamental to building Lightning Components apps. You start with, or build, simple, “fine-grained” components, where each component provides a defined set of self-contained functionality. Then you assemble those components into new components with higher-level functionality. And then you use *those* components, and “level up” again.

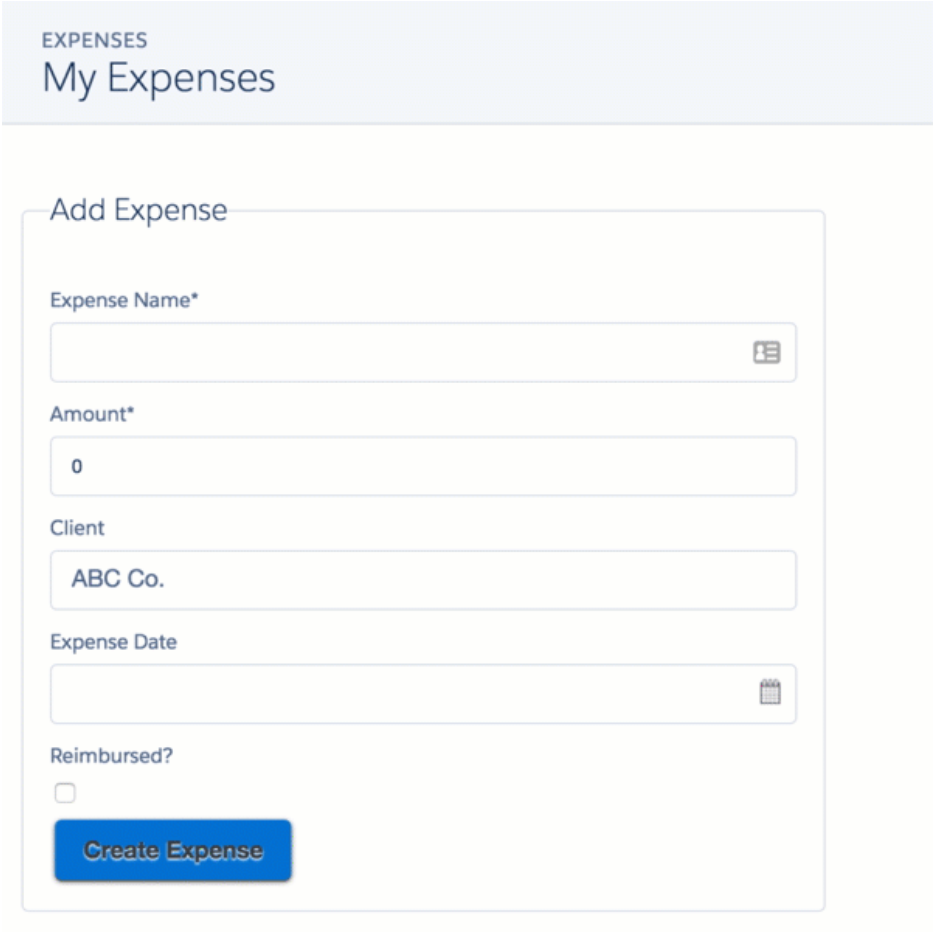
Let’s talk about that with a metaphor that’s familiar to us outside of the software context. Imagine a house. Better yet, let us show you one.

When you look at this house, what do you see? If you stop thinking of it as a “house”, but as a house *component*, you also start to see the pieces and patterns it’s made of.

At the largest scale, this house is composed of three similar structures. Those three *components* have similar, but not identical designs. Each can be broken down further, into even smaller components, such as windows, which can be broken down into individual panes. The arrangement, or composition, of these smaller components defines the differences between the three larger structures.

The three structures are joined together by two smaller, narrow structures/components, which can themselves be broken down into smaller, reusable patterns. These connecting components bring the three separate structures together into a larger whole: The House.

As we did with architecture, so we can do with a web app. Later in this module you'll take fine-grained input components and create a form component with them. Then you'll take that form component and put it into another component, to build up to app-level functionality.



It's not as pretty to look at as the house, but the principles behind the composition process are very similar. Thinking in terms of components, and composition, is a fundamental skill you'll develop throughout this module, and whenever you build apps with Lightning Components.

You've done a little bit of that here, but before we can really build components that *do something*, we need to learn about attributes, types, values, and expressions. Before that, though, your first code challenge!

Resources

Remember, this module is meant for Lightning Experience. When you launch your hands-on org, switch to Lightning Experience to complete this challenge.

Attributes and Expressions

Learning Objectives

After completing this unit, you'll be able to:

- Define attributes on your components, and pass attribute values into nested components.
- Understand the difference between a component definition and a component instance, and create multiple instances of a component.
- Create basic expressions to display changing and calculated values.

- Create conditional expressions for dynamic output.

Component Attributes

To this point, while we've created a couple of components, and learned a fair bit about building apps with them (at a high level), the code we've written isn't doing much more than what plain HTML does. That is, the two components we've created output the same static text, no matter what we do. You could put a dozen of them on the same screen, and they'd always say the same thing.

Boring.

To change that, we need to learn two things. First, we need to learn how to enable a component to accept input when it's created. That is, we need to set values on the component. We do this using attributes.

(The second thing we need to learn is how to actually *use* these values to change a component's behavior and output. We'll do that after we figure out attributes.)

Attributes on components are like instance variables in objects. They're a way to save values that change, and a way to name those value placeholders. For example, let's say we wanted to write a `helloMessage` component that prints a custom message. We can envision adding a message attribute to this component to customize its output. And then we can set that message when we add the component to our app, like the following.

```
<aura:component>

    <c:helloMessage message="You look nice
today."/>

</aura:component>
```

(You'll want to add this to your org, because we'll use it a few more times as we go. But if you do it now you'll get an error. Why is that? Because the `helloMessage` component doesn't exist yet. Lightning components validates your code as you write it. If you try to save code that it knows is invalid—for example, referencing a non-existent component—you'll get an error. So, let's figure out how to create `helloMessage` first.)

You can set a component's attributes when you create it, as we did in the preceding example. You can also change them over the course of your component's lifecycle, in response to actions the user takes, or events that happen elsewhere, and so on. And you can of course read and use attribute values in a number of different ways. We'll look at those when we get to expressions.

For the moment, let's look at how you define attributes for a component. An attribute is defined using an tag, which requires values for the `name` and `type` attributes, and accepts these optional attributes: `default`, `description`, `required`.

Whoa, that's a lot of different ways to use "attribute" in a sentence! It's really easy to get confused here, because we have three different concepts with similar names. Let's be specific.

1. A component attribute is the place where you can store a value. In the preceding example, the `helloMessage` component has a component attribute named `message`. Most of the time we're talking about component attributes.
2. You define a component attribute using the tag. We'll see an example of that momentarily. Let's call these attribute definitions.
3. The tag itself takes attributes when you use it! ☺ That is, when you define a component attribute using the tag, you set attributes on that specify the "shape" of the component attribute you're defining. ☹ Wait, let's try again: add a component attribute definition by setting attributes on the attribute's definition. 😭 A component attribute's attribute definition takes attributes? 😴

This is why writers 😲. Let's try to solve this terminology problem with some code. 😄

Here's the start of our `helloMessage` component:

```
<aura:component>

    <aura:attribute name="message"
type="String"/>

    <p>Hello! [ message goes here, soon ]</p>

</aura:component>
```

The `helloMessage` component has one component attribute, and that attribute is defined by setting the `name` and `type` of the attribute. The name of the attribute is `message` and, once we learn about expressions, that's how you'll reference it. It still only outputs static text and HTML, but we're inching closer to something useful.

✍ ?

The other attribute we've used here is `type`, and we've set it because it's required in an attribute definition. It says that the `message` attribute contains a string, which makes sense. We'll talk more about attribute data types, and the other parts of attribute definitions, but first let's learn about expressions,

and make `helloMessage` actually do something.

Expressions

Instead of getting lost in words again, let's dive right into making `helloMessage` work as intended.

```
<aura:component>

    <aura:attribute name="message"
type="String"/>

    <p>Hello! {!v.message}</p>

</aura:component>
```

Is that anticlimactic or what?

We're outputting the contents of `message` using the expression `{!v.message}`. That is, this expression references the `message` attribute. The expression is evaluated, and resolves to the text string that's currently stored in `message`. And that's what the expression outputs into the body of the component.

Ummm…what the heck is an "expression"?

An expression is basically a formula, or a calculation, which you place within expression delimiters ("`{!`" and "`}`"). So, expressions look like the following:

```
{!}
```

The formal definition of an expression is a bit intimidating, but let's look at and then unpack it: *An expression is any set of literal values, variables, sub-expressions, or operators that can be resolved to a single value.*

Yep, basically a formula, much like you'd write in a calculation field, filter criteria, or Visualforce. The formula, or expression, can contain a variety of things. Literal values should be obvious; they're things like the number `42`, or the string "`Hello`". Variables are things like the `message` attribute. Operators are things like `+`, `-`, and so on, and sub-expressions basically means you can use parenthesis to group things together.

Let's try this out, by making our expression slightly more complex.

```
<aura:component>

    <aura:attribute name="message"
type="String"/>

    <p>{!'Hello! ' + v.message}</p>

</aura:component>
```

All we've done is move the "Hello" part from static text outside the expression to literal text inside the expression. Notice that we've used the "+" operator to concatenate the two strings together. This might seem like a pretty small difference, but moving the greeting text inside the expression lets you use labels, instead of literal text, which makes it easier to update (and translate) your components. For example:

```
{!$Label.c.Greeting +
v.message}
```

Did you notice what our formal definition of expressions left out? JavaScript function calls. You can't use JavaScript in expressions in Lightning Components markup.

One last thing about expressions before we move on. You don't normally use a plain expression to output its result. Instead, you assign expressions to attributes of components, and let the component do the outputting. This gives us an opportunity to start looking at some of the built-in components provided with Lightning Components. Here's another version of our `helloMessage` component that uses to render our greeting.

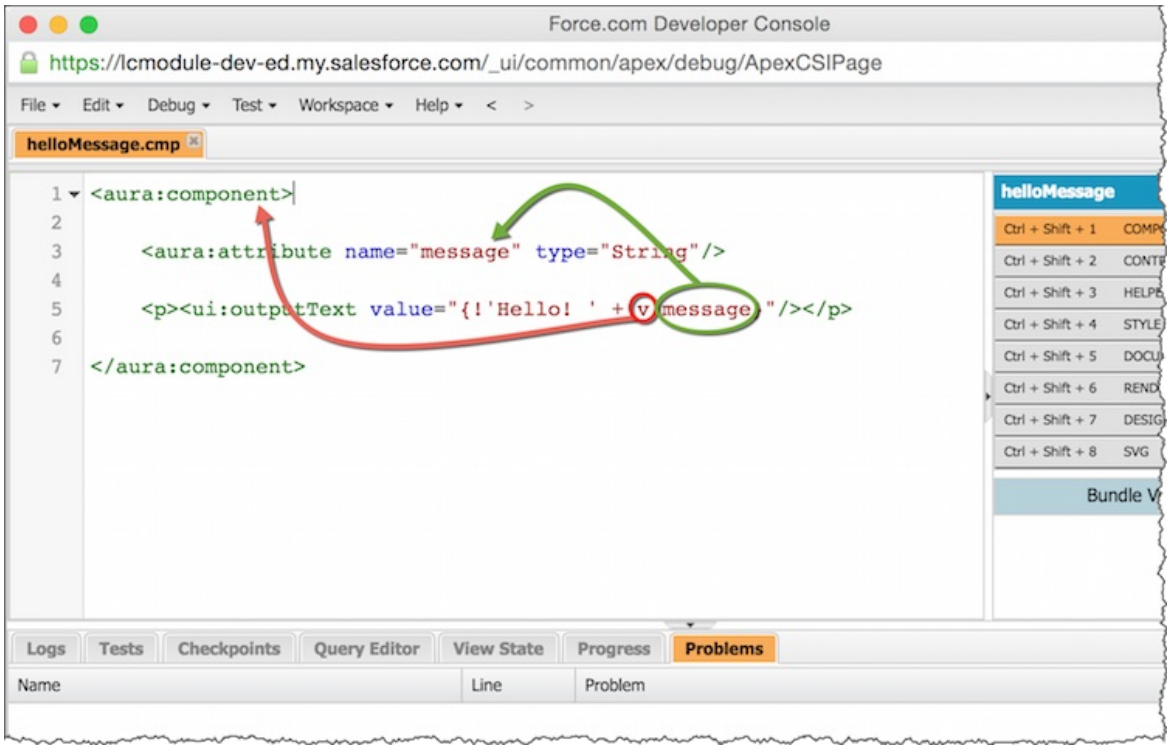
```
<aura:component>

    <aura:attribute name="message" type="String"/>

    <p><ui:outputText value="{!'Hello! ' + v.message}"/>
</p>

</aura:component>
```

Using has a few advantages over a bare expression. For one, the output value is escaped, making it safe for displaying user input, etc. For another, the output text is automatically wrapped in a tag, and you can attach a CSS class to that span, by setting the `class` attribute. is simple, but useful. We'll introduce more of these built-in components in our examples as we go.

Value Providers

Actually, we need to talk about another aspect of expressions. In the preceding examples, we've referenced the `helloMessage` component's message attribute with `v.message`. What's the "`v.`" part?

`v` is something called a value provider. Value providers are a way to group, encapsulate, and access related data. Value providers are a complicated topic, so for now, think of `v` as an automatic variable that's made available for you to use. In our component, `v` is a value provider for the view, which is the `helloMessage` component itself.



`v` gives you a "hook" to access the component's `message` attribute, and it's how you access *all* of a component's attributes.

Values in a value provider are accessed as named properties. To use a value, separate the value provider and the property name with a dot (period). For example, `v.message`, as we've seen.

When an attribute of a component is an object or other structured data (that is, not a primitive value), access the values on that attribute using the same dot notation. For example, `{!v.account.Id}` accesses the Id field of an account record. For deeply nested objects and attributes, continue adding dots to traverse the structure and access the nested values.

Attribute Data Types

Accessing structured data is a nice segue back to talking about attributes, and specifically about non-primitive attribute types. `message` is a string, but there are a number of different attribute types.

- Primitives data types, such as Boolean, Date, DateTime, Decimal, Double, Integer, Long, or String. The usual suspects in any programming language.
- Standard and custom Salesforce objects, such as Account or MyCustomObject__c.
- Collections, such as List, Map, and Set.
- Custom Apex classes.
- Framework-specific types, such as Aura.Component, or Aura.Component[]. These are more advanced than we'll get to in this module, but you should know they exist.

Here's a stripped down look at the `expenseItem` component, which we'll fill in later. It illustrates how you define an attribute for a custom object, and how to access fields on a record.

```
<aura:component>

    <aura:attribute name="expense" type="Expense__c"/>

    <p>Amount:
        <ui:outputCurrency value="{!v.expense.Amount__c}"/>
    </p>
    <p>Client:
        <ui:outputText value="{!v.expense.Client__c}"/>
    </p>
    <p>Date:
        <ui:outputDate value="{!v.expense.Date__c}"/>
    </p>
    <p>Reimbursed?:
        <ui:outputCheckbox value="
{!v.expense.Reimbursed__c}"/>
    </p>

</aura:component>
```

This component has one attribute, `expense`, which is the custom object we created waaaay back at the start of this module. The component's purpose is to display the details of an expense, and so we see it uses a number of different output components. Each of them outputs a single field from an Expense__c record, by referencing the field in an expression, `{!v.expense.`*`fieldName`*`}`.

Other Aspects of Attribute Definitions

When it comes to the attributes you set on the tag, here's the rest of what you need to know.

- The `default` attribute defines the default attribute value. It's used when the attribute is referenced and you haven't yet set the attribute's value.
- The `required` attribute defines whether the attribute is required. The default is `false`.
- The `description` attribute defines a brief summary of the attribute and its usage.

Setting the default value for an attribute with a complex data type can be kind of tricky. We'll see an example later, though, so for now we'll just give you the heads up.

Fun with Attributes and Expressions

To illustrate a few more concepts about attributes and expressions, let's create a really silly component, `helloPlayground`, with the following markup.

```
<aura:component>

    <aura:attribute name="messages" type="List"
        default="['You look nice today.',
            'Great weather we\'re having.',
            'How are you?']"/>


    <h1>Hello Playground</h1>

    <p>Silly fun with attributes and expressions.</p>


    <h2>List Items</h2>

    <p><c:helloMessage message="{!v.messages[0]}"/></p>
    <p><c:helloMessage message="{!v.messages[1]}"/></p>
    <p><c:helloMessage message="{!v.messages[2]}"/></p>


    <h2>List Iteration</h2>

    <aura:iteration items="{!v.messages}" var="msg">
        <p><c:helloMessage message="{!msg}"/></p>
    </aura:iteration>


    <h2>Conditional Expressions and Global Value
Providers</h2>

    <aura:if isTrue="{!$Browser.isIPhone}">
        <p><c:helloMessage message="{!v.messages[0]}"/></p>
    <aura:set attribute="else">
        <p><c:helloMessage message="{!v.messages[1]}"/></p>
        </aura:set>
    </aura:if>

</aura:component>
```

Now add the `helloPlayground` component to your harness app, and see how it runs!

There's a number of new things here. We won't go into depth on them right now, but you'll see all of these again.

First, `helloPlayground` has one attribute, `messages`, that's a complex data type, List. And it has a default value for that list, an array of three single-quoted strings separated by commas. And, in the List Items section, you can see how to access each of the strings using an index.

What happens if someone creates a with only two messages? Accessing the third item will fail, and while it won't cause a crash here, with more complex components it might.

So, in the List Iteration section, you can see a better way to work through all items in the list. The component repeats its body once per item in its `items` attribute, so the list shrinks or grows as we have fewer or more messages.

In the Conditional Expressions and Global Value Providers section, you can see a way to choose between two different possible outputs. The format is a bit awkward, because this is markup rather than, say, JavaScript, but the component lets you, for example, add an edit button to a page only if the user has edit privileges on the object.

Finally, something that is a little less obvious. In object-oriented programming, there's a difference between a *class* and an *instance* of that class. Components have a similar concept. When you create a .cmp resource, you are providing the *definition* (class) of that component. When you put a component tag in a .cmp, you are creating a *reference* to (instance of) that component.

It shouldn't be surprising that we can add multiple instances of the same component with different attributes. In the preceding example, when you use the default value for messages, you'll end up with eight references to (instances of) our component. If you pass in a longer list, you could end up with (many) more. All from our one little component!

And with that, friend, we think you're ready for some real work.

Resources

Remember, this module is meant for Lightning Experience. When you launch your hands-on org, switch to Lightning Experience to complete this challenge.

Handle Actions with Controllers

Learning Objectives

After completing this unit, you'll be able to:

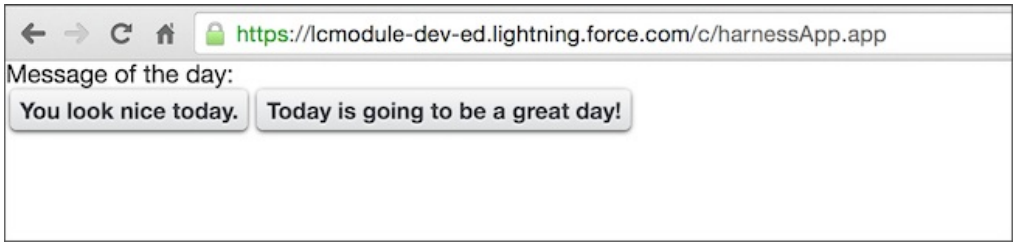
- Create a client-side controller to handle user actions.
- Read values from component attributes.
- Read values from user interface controls in your component.
- Write controller code in JavaScript that changes the user interface.

Handling Actions with Controllers

Up to now, we've worked only with XML-style markup. Up to now, the only way to have our component's output change was to change that markup. Up to now, our components didn't react to user input. Up to now, we haven't written any JavaScript.

That all changes in this unit.

To get started, let's look at a very simple component, and imagine what it needs to do to be able to handle its simple behavior.

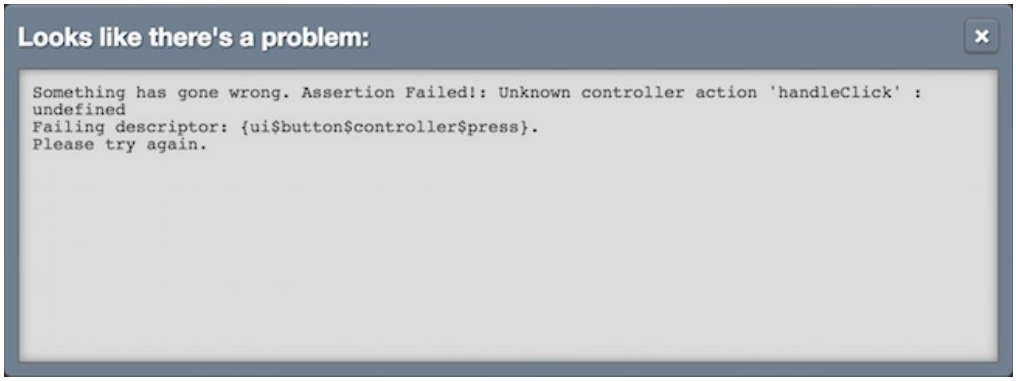


This is `helloMessageInteractive`, and it's hard to imagine a simpler component that "does something." It's a bit of static text, a (currently blank) message, and two buttons. Here's the code:

```
<aura:component>

    <aura:attribute name="message" type="String"/>

    <p>Message of the day: <ui:outputText value="{!v.message}"/>
</p>

    <div>
        <ui:button label="You look nice today."
            press="{!c.handleClick}"/>

        <ui:button label="Today is going to be a great day!"
            press="{!c.handleClick}"/>
    </div>

</aura:component>
```

This should look familiar. All we've really done is added two components to `helloMessage`. When you click a button, the message of the day is updated.

Well, not quite yet. If you've already entered the code and tried it yourself, you've noticed that when you click either of the buttons you get an error message.

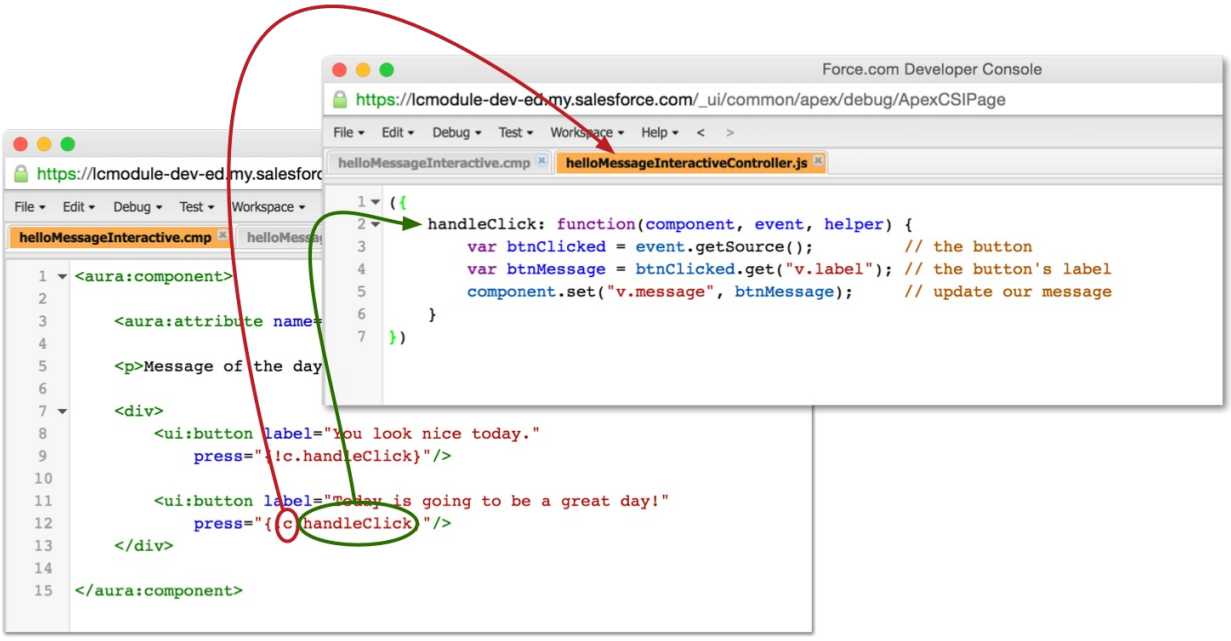


We'll be the first to admit that not every error message you'll see in Lightning Components is as helpful as you might hope. But this one is! It says that there's no controller action named "handleClick". Where does "handleClick" come from? It comes from the expression we assigned to the `press` attribute on each of the two tags:

```
press="{!c.handleClick}"
```

Given that this is a button, you can probably guess that the `press` attribute is how you assign a behavior to the button for when it's clicked. But what have we assigned? An expression, `{!c.handleClick}`, that's maybe a bit mysterious.

It's actually pretty simple. Just like the `v.message` expression from earlier, `c.handleClick` is a value provider, `c`, with a property, `handleClick`. `c` is the value provider for the component's client-side controller, and `handleClick` is a function defined in that controller. So, `{!c.handleClick}` is a reference to an action handler in the component's controller.



Uh, What's a Controller?

Whoops! A controller is basically a collection of code that defines your app's behavior when "things happen," whereby "things" we mean user input, timer and other events, data updates, and so on. If you look up "Model-View-Controller" on any number of developer sites, you'll get various definitions. For our purposes, for Lightning Components, a controller is a resource in a component bundle that holds the action handlers for that component. And action handlers are just JavaScript functions with a particular function signature.

Beyond the Basics

We talk a lot about controllers in this unit, and we know that the component itself is a view. We even mentioned the MVC, or Model-View-Controller, design pattern that's so common in web app frameworks. Is Lightning Components built on the MVC pattern?

In a word, no. There are similarities, to be sure, but it would be more correct to say that Lightning Components is View-Controller-Controller-Model, or perhaps View-Controller-Controller-Database.

Why is "controller" doubled up in that pattern name? Because when interacting with Salesforce, your components will have a *server-side* controller in

addition to the *client-side* controller we've worked with in this unit. This dual controller design is the key difference between Lightning Components and MVC.

What's the distinction between "model" and "database"? In traditional MVC, the model is a programmatic abstraction between the underlying data storage (usually a relational database) and the rest of the application. In Lightning Components, there's no Apex class that directly stands in between `@AuraEnabled` controller methods and DML operations. But then again, sObjects are already an abstraction between your Apex code and the underlying storage layer. You can add calculation fields, validation logic, and even add fully programmatic behavior in the form of triggers. So, is it a database or a model? We say po-TAY-tow but it's totally cool if you want to go with po-TAH-tow.

Confused? Excited? We'll get you sorted out on the details of server-side controllers in a later unit.

Let's look at the `helloMessageInteractive` controller in more detail, and make that explanation a bit more concrete.

```
({
    handleClick: function(component, event, helper) {
        var btnClicked = event.getSource();         // the button
        var btnMessage = btnClicked.get("v.label"); // the button's
label
        component.set("v.message", btnMessage);     // update our
message
    }
})
```

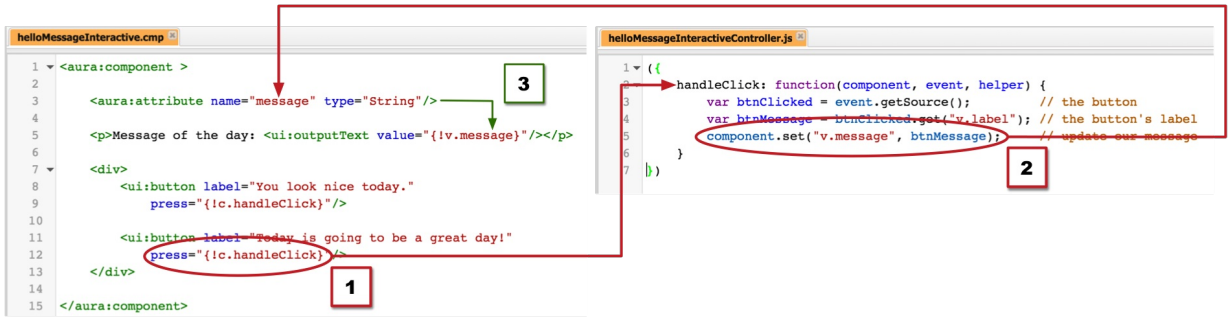
Controller resources have an interesting format. They are JavaScript objects that contain a map of name-value pairs, where the name is the name of the action handler and the value is a function definition.

Action Handlers

The combination of name-value pair and specific function signature is an action handler. You'll hear or see the terms action handler, controller action, and controller function used interchangeably, and for the most part that's correct. They almost always refer to the same thing. (We're not going to worry about the exceptions in this module.)

Don't worry too much about the special format of the controller resource. When you click the **CONTROLLER** button in the Developer Console, you'll get a controller resource with an example action handler already added. The one trick is—and you will get syntax errors if you forget—you need to put commas between action handlers. This is just basic JavaScript syntax, and we'll see the specifics in a bit.

The actual `handleClick` function is only four lines of code, but they might seem hard to understand at first. At a high level, it's simple: When the button is clicked, its action handler gets called (1). In the action handler, the controller gets the button that was clicked, pulls the label text out of it, and then sets the component's `message` attribute to that text (2). And the message of the day is updated (3). *You look nice today!*



Simple, right? Well…

Because this is super important, let's break it down line-by-line.

```
handleClick: function(component, event, helper)
{
```

The action handler name, followed by an anonymous function declaration. The important thing here is the function signature. While it's not technically required, you should *always* declare your controller functions to take these three parameters. We'll talk more about them as we go, but for now, these parameters represent:

- `component`—the component. In this case, it's `helloMessageInteractive`.
- `event`—the event that caused the action handler to be called.
- `helper`—the component's helper, another JavaScript resource of reusable functions.

```
    var btnClicked = event.getSource();         // the
button
```

Remember that `handleClick` is connected to our tag and its `press` attribute. The `event`, then, is someone clicking (**press**ing) the button. Inside that event it has the notion of a source, which is the button itself. So, calling `event.getSource()` gets us a reference to the specific that was clicked.

```
    var btnMessage = btnClicked.get("v.label"); // the button's
label
```

What do we do now that we have a reference to the button? We look inside it and get its label, which is set on the in the component markup. For example, .

Let's think about that a bit more. We don't have the definition of in front of us, but `label` is just another attribute, much like the `message` attribute we added to `helloMessageInteractive`. You can call `get()` on any component and provide the name of the attribute you want to retrieve, in the format `v.`*`attributeName`*. The result is the attribute value.

Note that, as in component markup, `v` represents the view, the component itself—but in this case, it's the child component, not `helloMessageInteractive`! Think of it this way. `btnClicked.get("v.label")` taps on the shoulder of whatever component `btnClicked` is and says "Hey, give me v.label". That component thinks "v is me," looks inside itself, and returns the value of its label attribute.

So now that we have a text string retrieved from the button, we just have one step left: to change our `message` attribute to the new message text. Unsurprisingly, just as `get()` *reads* a value from a component, `set()` *writes* a value.

```
    component.set("v.message", btnMessage);     // update our
message
```

However, let's notice one important difference. We called `get()` on `btnClicked`, the that's inside `helloMessageInteractive`. We're calling `set()` on `component`—the `helloMessageInteractive` component itself. This is a pattern you'll repeat in virtually every component you create: get values from child components, maybe do some processing, and set values in the component itself.

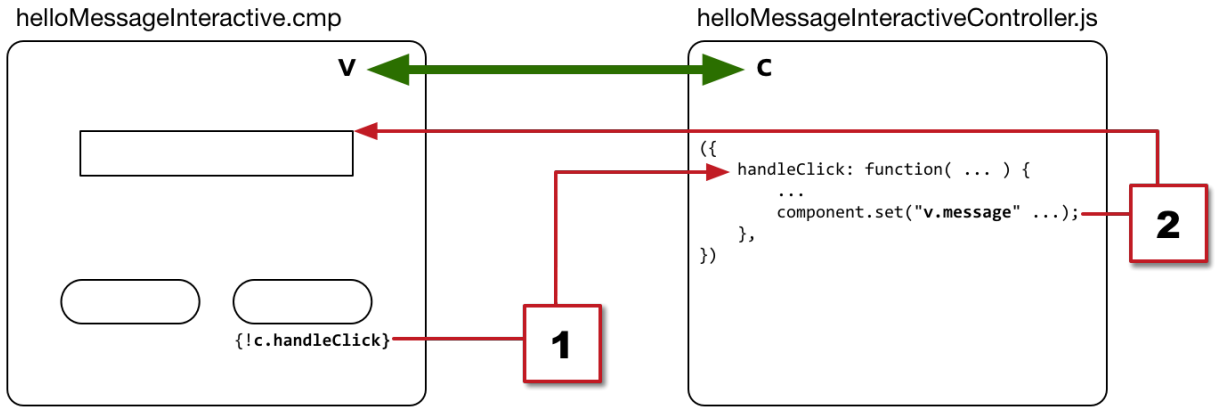
The Lightning Components View-Controller Programming Model

OK, gut check time. Is this making sense? For sure? If you just *think so*, make sure you've actually created the `helloMessageInteractive` component using the preceding code. It's one component, and copy/pasting the code takes two minutes, but being able to play with it is essential for understanding handling actions.

What you've done here might seem very simple, because it's not very many lines of code. But these lines of code illustrate some of the fundamental concepts of building apps with Lightning Components.

You can think of hooking up components to action handlers as "wiring them up." Think of `helloMessageInteractive` as a simple electrical circuit. There are switches, and there are light bulbs. (The Lightning Components framework supplies the power.) By itself, a switch might make a nice clicking sound, but until you wire it up, it's not very functional. You can have the trendiest Edison-style light bulb, but until you wire it up, it doesn't illuminate anything.

And so it is with Lightning Components. Way back ago, we said that the different resources in a component bundle are "auto-wired" to each other. And that's true: the wiring takes the form of the `v` and `c` value providers. They're automatically created and made available within your components, so your controller can reference the component and vice versa. But this auto-wiring takes place only at a high level—in `helloMessageInteractive`, between the component .cmp resource and the controller .js resource. This is the green arrow in the following illustration.



The wiring to hook up a specific component to a specific action handler—that is, wiring things that *generate* events, such as buttons, to the things that *handle* events, like a specific controller function—that's wiring you need to do yourself. Which, in fact, you just **did** yourself! These are the red arrows needed to complete a working circuit.

Adding `{!c.handleClick}` to the `press` attribute of a component (1) wires it up to the specific action handler. Calling `component.set("v.message", newMessage)` (2) wires up the result of that action handler to the component's `message` attribute. Which is itself wired up to a component for actual display, by referencing the `message` attribute in its value: .

You can further think of the `press` event as an electron flowing along the circuit you've created. If you haven't created a complete circuit, the event doesn't go anywhere, and nothing happens. When you start writing your own components, keep this in mind. Do you have a complete circuit? Are you sure? When in doubt, it sometimes helps to sketch it all on a whiteboard or paper, and confirm every connection.

You'll wire together components, events, and handlers, at low and high levels, so often you'll feel like an electrician. (Or, given the name of the framework, maybe Ben Franklin.) Wiring things together is *the* fundamental programming model in Lightning Components.

So, let's do it some more. After all, practice makes perfect.

Function Chaining, Rewiring, and Simple Debugging

Our first version of `handleClick` was three lines of code, because we broke each step in the get-process-set pattern into separate lines. You can use something called function chaining to collapse those into fewer lines. Since you'll probably see this in other Lightning Components code, let's give it a whirl ourselves. Add the following additional code to your `helloMessageInteractive` controller, after `handleClick`.

```
    handleClick2: function(component, event, helper) {
        var newMessage = event.getSource().get("v.label");
        component.set("v.message", newMessage);
    },

    handleClick3: function(component, event, helper) {
        component.set("v.message",
event.getSource().get("v.label"));
    }
```

Whoopsie! Did you just get a syntax error when trying to save? Remember earlier when we said you need to add commas between your action handlers? That's the issue here. Add a comma after the final brace ("`}`") of `handleClick`, as you can see at the end of `handleClick2` in the preceding snippet.

These action handlers do exactly the same thing as handleClick, in fewer lines of code. They do this by skipping intermediate variables, often by "chaining" directly to the next function call, by adding that call to the end of the previous one, separated by a period. (This concept is most clear when looking at the differences between `handleClick` and `handleClick2`.)

Which style you prefer is a matter of personal taste, and perhaps your organization's coding style. Your humble author prefers `handleClick2`, separating the `get` and `set`, but not bothering with a variable for the button, which we only need for its label text.

Of course, you can't verify that the new action handlers work until you wire a to use one of them, by setting the `press` attribute to `{!c.handleClick2}` or `{!c.handleClick3}`. Think of this as disconnecting the wires from one light bulb, and then attaching them to a different light bulb. It's that easy!

At that point, you reload the app, click one of the rewired buttons, and…well, it's the same by design, isn't it? How can we even tell which action handler is being called?

Sometimes simple is best. Let's add a bit of logging to one of the action handler functions:

```
    handleClick2: function(component, event, helper) {
        var newMessage = event.getSource().get("v.label");
        console.log("handleClick2: Message: " +
newMessage);
        component.set("v.message", newMessage);
    },
```

Now if you wire up a to `handleClick2`, you'll see a log message in your browser's JavaScript console whenever you click it.

Yes, there are more sophisticated debugging tools, but printing things to the console is a time-honored debugging technique. If you want to output an object of some sort, wrap it with `JSON.stringify(yourObject)` to get even more useful details. When you're after a quick peek, `console.log()` is your friend.

We won't cover those more sophisticated debugging tools and techniques here, but see Resources for some great tools and instructions.

OK, `helloMessageInteractive` was stupid simple and has a hard-coded (and relentlessly positive) attitude. In the next unit, we'll work with something more complex, and learn how to capture real user input. And since people in the real world aren't always so positive, we'll learn how to validate their input,

too.

Resources

Remember, this module is meant for Lightning Experience. When you launch your hands-on org, switch to Lightning Experience to complete this challenge.

Input Data Using Forms

Learning Objectives

After completing this unit, you'll be able to:

- Create a form to display current values and accept new user input.
- Read values from form elements.
- Validate user input and display error messages for invalid input.
- Refactor code from a component's controller to its helper.

Input Data Using Forms

As of this unit, we're done with `helloWhatever`-style components. From here, we'll be creating and assembling the expenses tracker mini-app that we previewed earlier. We'll spend the majority of this unit building and understanding the form that lets you create a new expense.

The expenses App Container

But before we get started with that, let's also be done with creating plain, or downright ugly, components. The first thing we'll do, then, is pull in the Salesforce Lightning Design System, or SLDS, and "activate" it in our app. The way we'll do this lets us talk a little more about app containers.

Note

We're not actually going to discuss SLDS itself in this unit, or anywhere in the rest of the module. We're going to focus here on getting it added to an app, and then within our example code we'll use the SLDS classes, but not explain them in detail. See Resources for *lots* of different ways to learn more about SLDS.

Today, SLDS is automatically available to your components when they run inside Lightning Experience or Salesforce1. We sometimes call this running in the one.app container. This built-in version is the same that's used by many of the standard Lightning components. However, SLDS is *not* available by default in a stand-alone app, or when you use your components in Lightning Out or Lightning Components for Visualforce. These are different app containers, and they provide different services and resources. We'd like to create our expense app in such a way that it works and looks good in all of these contexts. And fortunately it's not actually that hard to do.

The way we'll do it is by adding SLDS to our harness application. Then, inside the "real" expenses app (really, the top-level component, and all its child components), we can use SLDS tools and techniques, without worrying about where the SLDS resources—style sheets, icons, and so on—come from. That is to say, our app container (the harness app) sets up resources inside its context so that any app running *inside* that container has the resources it needs.

Let's convert those wordy concepts to some code. Create a new expensesApp.app Lightning application with the following markup.

```
<aura:application extends="force:slds">

        <!-- This component is the real "app" -
->
        <!-- c:expenses/ -->

</aura:application>
```

Here's what's going on. The `extends="force:slds"` attribute activates SLDS in this app, by including the same Lightning Design System styles provided by Lightning Experience and Salesforce1. But note that this harness app is just a wrapper, a shell. The *real* app is the `expenses` component, which we haven't created. (That's the part; it's commented out because we can't save our app until the `expenses` component actually exists.)

By way of the wrapper app, our components use the `extends="force:slds"` mechanism to get access to SLDS *when they run from this application*. When they run inside Lightning Experience or Salesforce1, with no code changes, they use that container's automatic inclusion of SLDS.

In this case, that amounts to the same thing. But this concept of using the outer harness app to set up a context so that the real app doesn't need to worry about context differences isn't limited to style resources. You can use this to provide replacement event handlers, for example…although that's getting ahead of ourselves. Let's learn to walk before we try to fly!

The expenses App Component

The next step is to create the component that's the top level for our expenses app. (Remember, even though we call it an "app," it's really just another Lightning component.) In the Developer Console, create a new Lightning component named "expenses", and replace the default markup with the following.

```
<aura:component>

    <!-- PAGE HEADER -->
    <div class="slds-page-header" role="banner">
      <div class="slds-grid">
        <div class="slds-col">
          <p class="slds-text-heading--label">Expenses</p>
          <h1 class="slds-text-heading--medium">My
Expenses</h1>
        </div>
      </div>
    </div>
    <!-- / PAGE HEADER -->

    <!-- NEW EXPENSE FORM -->
    <div class="slds-col slds-col--padded slds-p-top--large">


        <!-- [[ expense form goes here ]] -->


    </div>
    <!-- / NEW EXPENSE FORM -->

</aura:component>
```

Now you can uncomment the tag in the actual .app, and open the preview of what is for now just an empty shell. You should see something like the following.

Not a lot going on yet, but it's kind of exciting to see the SLDS styling already having an effect. Remember that we're not going to explain most of the SLDS markup, but we'll include comments in the markup. You can see how we created the header for the app, and start to get the idea.

The New Expense Form

Before we start on the form, let's just acknowledge something upfront: What we're about to do is temporary. Remember all that talk talk talk about decomposing your app down into separate, smaller components, and then building up from there? We're not doing that here—not yet—and frankly it's a bit of a cheat.

But it's a cheat in service to keeping the code from getting too complicated, too quickly. We're doing it this way so we can focus on one lesson at a time. And, this isn't a bad way for you to work on things yourself: build inside one component until it gets too "busy," and then refactor and decompose into smaller sub-components. As long as you remember to refactor!

OK, . In `expenses` component, replace the comment with the following code for the Add Expense form.

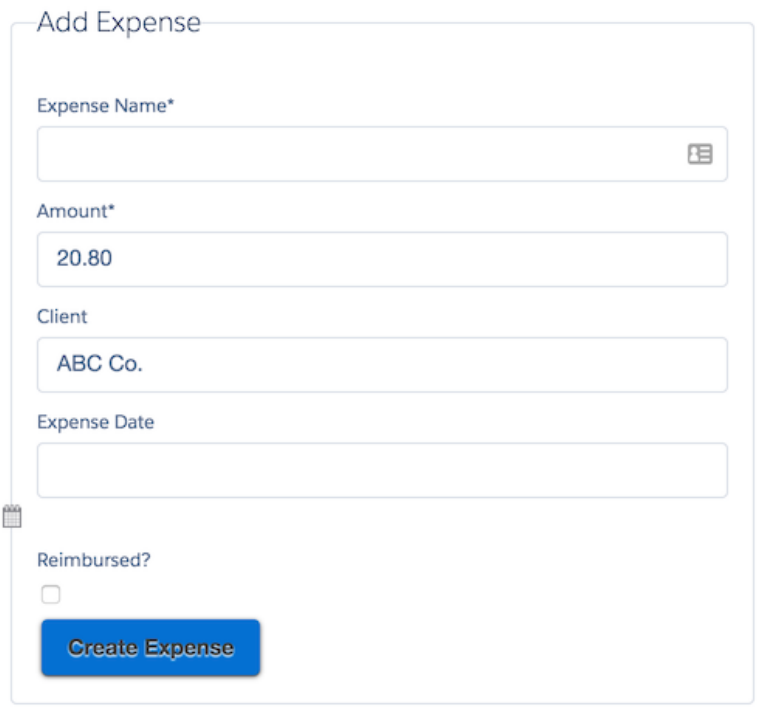
```
    <div aria-labelledby="newexpenseform">

    <!-- BOXED AREA -->
    <fieldset class="slds-box slds-theme--default slds-container--
small">

      <legend id="newexpenseform" class="slds-text-heading--small
        slds-p-vertical--medium">
        Add Expense
      </legend>

      <!-- CREATE NEW EXPENSE FORM -->
      <form class="slds-form--stacked">

        <div class="slds-form-element slds-is-required">
            <div class="slds-form-element__control">
                <ui:inputText aura:id="expname" label="Expense Name"
                    class="slds-input"
                    labelClass="slds-form-element__label"
                    value="{!v.newExpense.Name}"
                    required="true"/>
            </div>
       </div>

       <div class="slds-form-element slds-is-required">
            <div class="slds-form-element__control">
                <ui:inputNumber aura:id="amount" label="Amount"
                    class="slds-input"
                    labelClass="slds-form-element__label"
                    value="{!v.newExpense.Amount__c}"
                    required="true"/>

            </div>
        </div>

        <div class="slds-form-element">
            <div class="slds-form-element__control">
                <ui:inputText aura:id="client" label="Client"
                    class="slds-input"
                    labelClass="slds-form-element__label"
                    value="{!v.newExpense.Client__c}"
                    placeholder="ABC Co."/>
            </div>
        </div>

        <div class="slds-form-element">
            <div class="slds-form-element__control">
                <ui:inputDate aura:id="expdate" label="Expense Date"
                    class="slds-input"
                    labelClass="slds-form-element__label"
                    value="{!v.newExpense.Date__c}"
                    displayDatePicker="true"/>
            </div>
        </div>

        <div class="slds-form-element">
            <ui:inputCheckbox aura:id="reimbursed" label="Reimbursed?"
                class="slds-checkbox"
                labelClass="slds-form-element__label"
                value="{!v.newExpense.Reimbursed__c}"/>
        </div>

        <div class="slds-form-element">
            <ui:button label="Create Expense"
                class="slds-button slds-button--brand"
                press="{!c.clickCreateExpense}"/>
        </div>

      </form>
      <!-- / CREATE NEW EXPENSE FORM -->

    </fieldset>
    <!-- / BOXED AREA -->
```

```
  </div>
  <!-- / CREATE NEW EXPENSE -->
```

That looks like a lot of code to grasp at once. It's not. When you strip away the SLDS markup and classes, from a Lightning Components view this form boils down to just this:

```
<ui:inputText aura:id="expname" label="Expense Name"
    value="{!v.newExpense.Name}" required="true"/>

<ui:inputNumber aura:id="amount" label="Amount"
    value="{!v.newExpense.Amount__c}" required="true"/>

<ui:inputText aura:id="client" label="Client"
    value="{!v.newExpense.Client__c}" placeholder="ABC Co."/>

<ui:inputDate aura:id="expdate" label="Expense Date"
    value="{!v.newExpense.Date__c}" displayDatePicker="true"/>

<ui:inputCheckbox aura:id="reimbursed" label="Reimbursed?"
    value="{!v.newExpense.Reimbursed__c}"/>

<ui:button label="Create Expense" press="
{!c.clickCreateExpense}"/>
```

Here's the resulting form.



It's just five input components, and a button. We already know about buttons, so let's talk about some new elements of these input components.

Note

Just because we're not explaining the SLDS markup doesn't mean it's not important. Try creating the form with this simplified markup. It's hideous and unusable. SLDS rocks. We're just not *teaching* it here. (Go on, tell us this module isn't already long!)

First, notice that we've got a variety of input components, and that they're named in such a way to suggest that they go best with specific data types. That is, you're not surprised that you'd want to use with a date field, and so on. There's a range of different input components, well beyond these four, and it's always best to match the component type to the data type. The reason might not be obvious yet, but it will be when you try this app out on a phone—type-specific components can provide input widgets best suited to the form factor. For example, the date picker is optimized for mouse or finger tip, depending

on where you access it.

Next, notice that each input component has a label set on it, and that the `label` text is automatically displayed next to the input field. There are a few other attributes we haven't seen before: `required`, `placeholder`, `displayDatePicker`. If you can't guess what these are for, you can look them up. (And we'll come back to that deceptive `required`.)

Next, there's an `aura:id` attribute set on each tag. What's that for? It sets a (locally) unique ID on each tag it's added to, and that ID is how you'll read values out of the form fields. We'll look at how to do that very shortly.

Attributes for Salesforce Objects (sObjects)

But first we need to look at the `value` attribute. Each tag has a value on it, set to an expression. For example, `{!v.newExpense.Amount__c}`. From the format of the expression, you should be able to deduce a few things.

- `v` means this is a property of the view value provider. That means that this is an attribute on the component. (Which we haven't created yet.)
- Based on the dot notation, you can tell that `newExpense` is some kind of structured data type. That is, `newExpense` itself has properties. Or...fields?
- From the "__c" that's at the end of most of the property names, you can guess that these map back to custom fields, most likely on the Expense custom object.
- So, `newExpense` is probably an Expense object!

Cool, we haven't discussed this yet! Here's the actual attribute definition, which you should add to the top of the component, right after the opening tag.

```
    <aura:attribute name="newExpense"
type="Expense__c"
         default="{ 'sobjectType': 'Expense__c',
                        'Name': '',
                        'Amount__c': 0,
                        'Client__c': '',
                        'Date__c': '',
                        'Reimbursed__c': false }"/>
```

What's going on here is actually pretty simple. The name attribute you already know. And the type is, unsurprisingly, the API name of our custom object. So far, so good.

The default attribute isn't new, but the format of its value is. But it shouldn't be too hard to grasp. It's a JSON representation of an `sObject`, specifying the type of the object (again, the API name), and values for each of the fields to set on it by default. In this case, we're basically setting everything to a representation of an empty value.

And that's, well, most of what you need to know about sObjects! From here out, the Lightning Components framework will let you treat `newExpense`, in JavaScript and in markup, like it's a record from Salesforce—even though we're not (yet) loading it from Salesforce!

Handle Form Submission in an Action Handler

So we have a form. Right now, if you fill it out and click the button to create a new expense, what happens? Unless you've run ahead and created it already, you'll get another error about a missing controller action. This is because neither the controller, nor the action handler specified on the , have been created.

In the Developer Console, click the **CONTROLLER** button for the `expenses` component to create the controller resource. Then replace the default code with the following.

```
({
    clickCreateExpense: function(component, event, helper) {

        // Simplistic error checking
        var validExpense = true;

        // Name must not be blank
        var nameField = component.find("expname");
        var expname = nameField.get("v.value");
        if ($A.util.isEmpty(expname)){
            validExpense = false;
            nameField.set("v.errors", [{message:"Expense name can't be
blank."}]);
        }
        else {
            nameField.set("v.errors", null);
        }

        // ... hint: more error checking here ...

        // If we pass error checking, do some real work
        if(validExpense){
            // Create the new expense
            var newExpense = component.get("v.newExpense");
            console.log("Create expense: " + JSON.stringify(newExpense));
            helper.createExpense(component, newExpense);
        }
    }
})
```

OK, this is all new, so let's look at it carefully. First, let's note that this action handler function is basically divided into three sections, or steps:

1. Setup
2. Process form values
3. If there are no errors, do something

This structure might be familiar to you, because it's a pretty fundamental way to process user input in a web application. Let's look at each of the steps, to see how they work in Lightning Components.

For setup, all we do is initialize the state of our error checking. It's a simple flag, is this a valid expense? Each time the `clickCreateExpense` action handler is called, we'll start on the assumption that the expense data is OK, and then invalidate it if we find a problem.

Processing the form values is where things get interesting. For the moment we're only validating the expense name field, but there's a lot going on.

First, we're using a new function, `find()`, to get a reference to the component for the expense name. `find()` takes a single argument, an ID, and that ID corresponds to the `aura:id` attribute we set on the component.

Back in `helloMessageInteractive`, we didn't use `find()` to figure out the label text of the button that was clicked. This is because we didn't need to. We were able to get a reference to that button directly, by pulling it out of the `event` parameter using `event.getSource()`. You don't always have that luxury; indeed, it's rare when everything you need from user input comes only from the event.

So, when your controller needs a way to get to a child component, first set an `aura:id` on that component in markup, and then use `component.find(theId)` to get a reference to the component at runtime.

Note

`component.find()` only lets you access direct child components. It isn't a magic way to go wandering around the component hierarchy and read or change things. Remember, components are supposed to be self-contained, or to communicate with…well, we'll get to that.

Once we have a reference to the expense name field, getting the value of the user input is done using the familiar `nameField.get("v.value")`. Note that you can use function chaining to get a field's value using, for example, `component.find("expname").get("v.value")`. Here, since we actually need the reference to the field itself, we separate it into two steps.

Once we have the form value in a local variable, we can perform all manner of validation on it. In this case, we just want to ensure that it's not blank. To do so we're using a convenience utility object, `$A.util`, that's a part of the Lightning Components framework. `$A.util` has some nifty functions, but you

can do your validation with plain JavaScript, too.

It's when validation fails that things get interesting again. When the user enters invalid input, we want two things to happen:

1. Don't try to create the expense.
2. Show a helpful error message.

For the first, we set the `validExpense` flag to false. For the second, we make use of our reference to the child input component to set its `errors` attribute to a message that will help someone fill out the form correctly.

The specific format of the value we're setting on `errors` is an array of objects containing name-value pairs. The complexity of this (as opposed to a simple string) is a hint that there's a lot more you can do with and error handling. We unfortunately don't have the space here to go into it in depth. For now, make a note of the format.

By contrast, if the field passes validation, we set the field's component's errors attribute to null. That, at least, is straightforward, and it resets any errors that were previously found for that field.

And with that, we're on to step three in handling the form submission: actually creating the expense! As you can see, to prepare for that, we're getting the complete `newExpense` object from the component attribute: `component.get("v.newExpense")`. This gives us a single variable that we can use to create a new expense record.

Before we get to that, though, here's a question for you to think about: why don't we pull the form values out of `newExpense`? Get the structured variable once, at the start of the action handler, and then just access its properties, instead of what could be a long series of `find().get()` calls?

The reason is simple: because we need references to the individual fields to be able to set `errors` on them. It's also a good practice to validate the raw form data; your validation logic doesn't know what kinds of processing might happen within the `newExpense` object.

Create the New Expense

Remember earlier we said that putting the expense form in the main component was a bit of a cheat? Well, this next section doesn't get the "a bit" qualifier. What we're doing here is just flat out avoiding the complexity of really creating the record. And we're avoiding it now because *that's the entire next unit*. So, let's wrap this one up with something simple, but which still shows us some important concepts.

First, let's create a place to "store" new expenses. We'll simply create a local-only array of expenses to hold them. At the top of `expenses` component markup, right before the `newExpense` attribute, add a new `expenses` attribute, which will hold an array of expense objects.

```
    <aura:attribute name="expenses"
type="Expense__c[]"/>
```

All we need to do is update the `expenses` array. Which, it turns out, is easy (in the *cheater-cheater* version of our form), and illustrates yet another important concept.

In our controller, we've hidden the work of actually creating the new expense behind this function call: `helper.createExpense(component, newExpense)`. In software development, another word for "hiding" is *abstraction*. And we're using something called a helper to abstract away our cheating.

We discussed helpers briefly earlier, and we're not going to cover advanced details of helpers in this module. For now, let's say three things about helpers:

- A component's helper is the appropriate place to put code to be shared between several different action handlers.
- A component's helper is a great place to put complex processing details, so that the logic of your action handlers remains clear and streamlined.
- Helper functions can have any function signature. That is, they're not constrained the way that action handlers in the controller are. (Why is this? Because *you* are calling the helper function directly from your code. By contrast, the framework calls action handlers via the framework runtime.) It's a convention and recommended practice to always provide the component as the first parameter to helper functions.

OK, let's get on with it. In the Developer Console, click the **HELPER** button for the `expenses` component to create the associated helper resource, and then replace the example code with the following.

```
({
    createExpense: function(component, expense) {
        var theExpenses = component.get("v.expenses");

        // Copy the expense to a new object
        // THIS IS A DISGUSTING, TEMPORARY HACK
        var newExpense =
JSON.parse(JSON.stringify(expense));

        theExpenses.push(newExpense);
        component.set("v.expenses", theExpenses);
    }
})
```

For the moment, ignore the disgusting hack part. The other three lines of code illustrate a common pattern we've seen before, and which you'll use over and over: get-process-set. First we get the array of expenses from the `expenses` attribute. Then we add the new expense "record" to it. Then we update (`set`) the `expenses` attribute with the changed array.

The Reference Is Not the Collection

What's new here is that, for the first time, we're updating a collection, an array. And if you're an experienced programmer, you're probably wondering: "Why do I need the `set()` here?"

That is, `component.get("v.expenses")` gets a reference to the array stored in the `component` attribute. `component.set("v.expenses", theExpenses)` simply sets the `component` attribute to the same reference. Sure, in between, the contents of the array has been added to, but the container is the same: *the reference to the array hasn't actually changed!* So, why update it?

If you're having trouble understanding what this means, add two logging statements before and after the critical statements, and dump the contents of `theExpenses` to the console.

```
console.log("Expenses before 'create': " +
JSON.stringify(theExpenses));
theExpenses.push(newExpense);
component.set("v.expenses", theExpenses);
console.log("Expenses after 'create': " + JSON.stringify(theExpenses));
```

Reload and run the app, add at least two expenses, and look at the structure of `theExpenses`. Now comment out the `component.set()` line, and do it again.

What the...? `component.set()` doesn't affect `theExpenses` at all! But! But! But? What does it actually *do*?!?

You are absolutely correct to be asking this question. And the answer is: magic!

What `component.set()` does here isn't update the value of the `expenses` attribute. It triggers notification that the `expenses` attribute has changed.

The effect is that, anywhere in your app that you've referenced the `expenses` attribute in an expression, the value of that expression is updated, and that update cascades everywhere the `expenses` attribute was used. And they all update to rerender based on the new contents. This all happens behind the scenes, handled by the Lightning Components framework, as part of the auto-wiring that happens when you use `{!v.expenses}`. In a word, magic.

To summarize, if this were plain JavaScript, you wouldn't need the `component.set()`. In order to trigger underlying effects built into Lightning Components, you do. If you ever write some controller or helper code, test it, and nothing happens, make sure you've done the required `component.set()`.

The "disgusting hack" works around a similar issue with references. To see the issue, change the line to remove the two JSON calls, and test the app. You'll see what the problem is quickly enough. We'll remove it in the next unit, and so won't explain further.

Displaying the List of Expenses

So, for all that talk about "magically" updating anything that uses `{!v.expenses}`, guess what. Nothing else is using it, yet. Let's fix that.

In the Developer Console, create a new Lightning component named `expenseItem`, and replace the default markup with the following. If you already created `expenseItem`, just update the markup.

```
<aura:component>

    <aura:attribute name="expense" type="Expense__c"/>

    <p>Amount:
        <ui:outputCurrency value="{!v.expense.Amount__c}"/>
    </p>
    <p>Client:
        <ui:outputText value="{!v.expense.Client__c}"/>
    </p>
    <p>Date:
        <ui:outputDate value="{!v.expense.Date__c}"/>
    </p>
    <p>Reimbursed?:
        <ui:outputCheckbox value="
{!v.expense.Reimbursed__c}"/>
    </p>

</aura:component>
```

In the Developer Console, create a new Lightning component named `expensesList`, and replace the default markup with the following.

```
<aura:component>

    <aura:attribute name="expenses" type="Expense__c[]"/>

    <div class="slds-card slds-p-top--medium">
        <header class="slds-card__header">
            <h3 class="slds-text-heading--small">Expenses</h3>
        </header>

        <section class="slds-card__body">
            <div id="list" class="row">
                <aura:iteration items="{!v.expenses}"
var="expense">
                    <c:expenseItem expense="{!expense}"/>
                </aura:iteration>
            </div>
        </section>
    </div>

</aura:component>
```

There's not a lot new for you here. This is a component that displays a list of expenses. It has one attribute, `expenses`, which is an array of expense `(Expense__c)` objects. And it uses an to create an for each of those expense objects. The overall effect is, as we noted, to display a list of expenses. So far, so good.

Now add the `expensesList` component to the end of `expenses` component. Add it right before the ending tag in `expenses.cmp`.

```
<c:expensesList expenses="{!v.expenses}"/>
```

If you reload the application, you'll see an Expenses section below the form. (It's not quite right, visually, but it's good enough for now.)

What did we just do? We added the `expensesList` component to *and passed the main `expenses` attribute into it*. So now the instance of `expenses` that `expensesList` has is the same as the instance of `expenses` that the `expenses` component has. They are references to the same array of records, and through the magic of Lightning Components, when the main `expenses` array is updated, the `expensesList` component will "notice" and rerender its list. Give it a whirl!

Whew! That was a long unit, and it calls for a stretch break. Please, do get up and walk around for a few minutes.

Then come back, and we'll show you how to save your new expenses for real.

Resources

Remember, this module is meant for Lightning Experience. When you launch your hands-on org, switch to Lightning Experience to complete this challenge.

Connect to Salesforce with Server-Side Controllers

Learning Objectives

After completing this unit, you'll be able to:

- Create Apex methods that can be called remotely from Lightning Components code.
- Make calls from Lightning Components to remote methods.
- Handle server responses asynchronously using callback functions.
- Stretch goal: explain the difference between "c.", "c:", and "c.".

Server-Side Controller Concepts

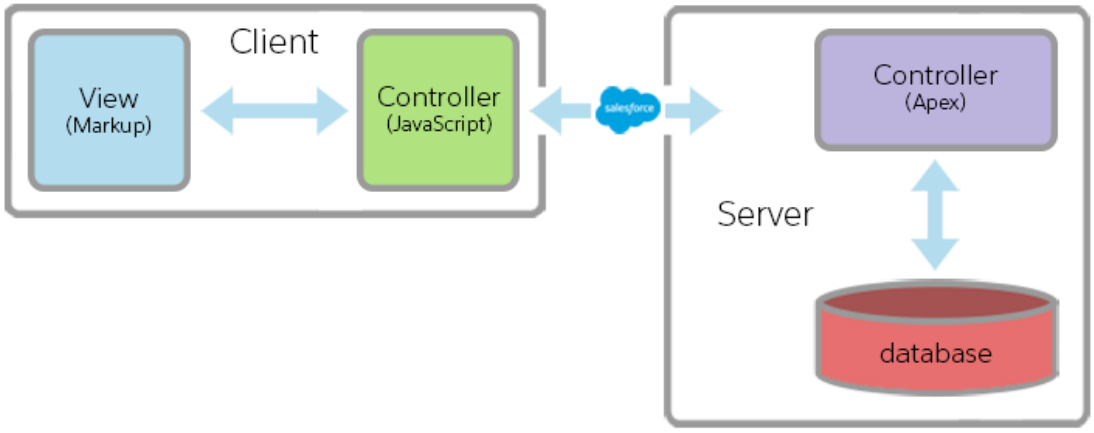
So far everything we've done has been strictly client-side. We're not saving our expenses back to Salesforce yet. Create a few expenses and then hit reload and what happens? That's right, all the expenses disappear. Woohoo, free money!

Except, Accounting just called and, well, they're kind of humorless about that kind of thing. And, actually, don't we want to be reimbursed for those expenses, which are otherwise coming out of *our* pocket...? Yikes! We need to save our data to Salesforce, no more delays!

Kidding aside, it's finally time to add server-side controllers to our app. We've been holding this back while we got the basics down. Now that you're ready for it, let's dive in!

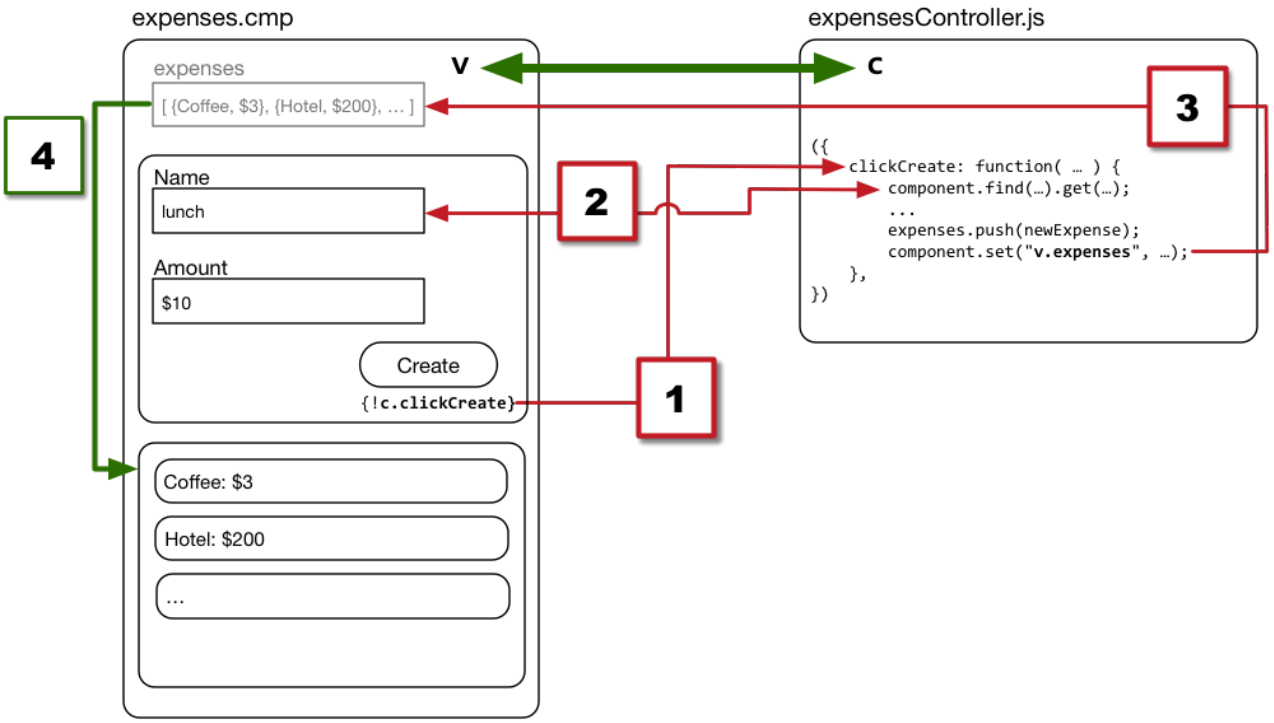
Into some pictures, that is. Let's make sure we know where we're headed, and that our gas tank is full, before we hit the road.

First, let's revisit the first diagram we saw in this module, a (very) high level look at the architecture of Lightning Components apps.



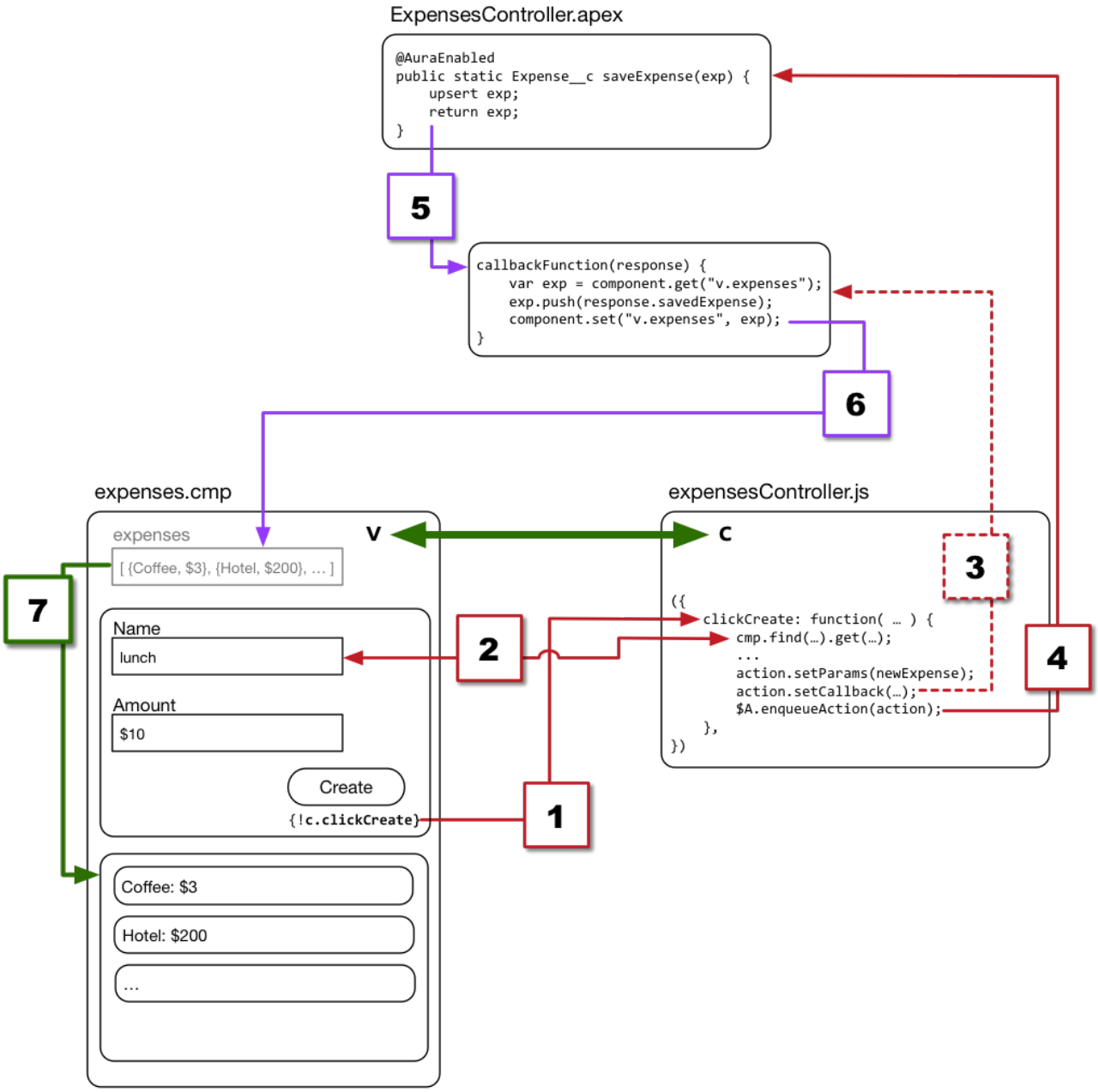
Up to now, everything we've looked at has been on the client side of this picture. (And note we simplified by merging controllers and helpers here.) Although we've referenced a custom object type, Expense__c, which is defined on the server side, we've never actually touched the server directly.

Remember how we talked about wiring together different elements to create a complete circuit? The expenses form that we built in the last unit might look something like this.



The circuit starts with the **Create** button, which is wired to the `clickCreate` action handler (1). When the action handler runs, it gets values out of the form fields (2) and then adds a new expense to the `expenses` array (3). When the array is updated via `set`, it triggers the automatic rerendering of the list of expenses (4), completing the circuit. Simple, right?

Well, when we wire in server-side access, the diagram gets a bit more complicated. More arrows, more colors, more numbers! (We'll hold off on explaining all of it for the moment.)



What's more, this circuit doesn't have the same smooth, synchronous flow of control. Server calls are expensive, and can take a bit of time. Milliseconds when things are good, and long seconds when the network is congested. Lightning Components doesn't want apps to be locked up while waiting for server responses.

The solution to staying responsive while waiting is that server responses are handled *asynchronously*. What this means is, when you click the Create Expense button, your client-side controller fires off a server request and then keeps processing. It not only doesn't wait for the server, it forgets it ever made the request!

Then, when the response comes back from the server, code that was packaged up with the request, called a *callback function*, runs and handles the response, including updating client-side data and the user interface.

If you're an experienced JavaScript programmer, asynchronous execution and callback functions are probably your bread and butter. If you haven't worked with them before, this is going to be new, and maybe pretty different. It's also *really* cool.

Querying for Data from Salesforce

We're going to start with reading data from Salesforce, to load the list of existing expenses when the Expenses app starts up.

Note

If you haven't already created a few real expense records in Salesforce, now would be a good time. Otherwise, after implementing what follows, you might spend time debugging why nothing is loading, when there's actually just nothing to load. Your humble author falls for this one All. The. Time.

The first step is to create your Apex controller. Apex controllers contain remote methods your Lightning components can call. In this case, to query for and receive expenses data from Salesforce.

Let's take a look at a simplified version of the code. In the Developer Console, create a new Apex class named "ExpensesController" and paste in the following code.

```
public with sharing class ExpensesController {

    // STERN LECTURE ABOUT WHAT'S MISSING HERE COMING SOON

    @AuraEnabled
    public static List<Expense__c> getExpenses() {
        return [SELECT Id, Name, Amount__c, Client__c,
Date__c,
                       Reimbursed__c, CreatedDate
                FROM Expense__c];
    }
}
```

We'll go into Apex controllers in some depth in the next section, but for now this is really a very straightforward Apex method. It runs a SOQL query and returns the results. There are only two specific things that make this method available to your Lightning Components code.

- The `@AuraEnabled` annotation before the method declaration.

  "Aura" is the name of the open source framework at the core of Lightning Components. You've seen it used in the namespace for some of the core tags, such as . Now you know where it comes from.

- The `static` keyword. All `@AuraEnabled` methods must be static methods, and either `public` or `global` scope.

If these requirements remind you of remote methods for Visualforce's JavaScript remoting feature, that's not a coincidence. The requirements are the same, because the architecture is very similar at key points.

One other thing worth noting is that the method doesn't do anything special to package the data for Lightning Components. It just returns the SOQL query results directly. The Lightning Components framework handles all the marshalling/unmarshalling work involved in most situations. Nice!

Loading Data from Salesforce

The next step is to wire up the `expenses` component to the server-side Apex controller. This is so easy it might make you giddy. Change the opening tag of the expenses component to point at the Apex controller, like this.

```
<aura:component controller="ExpensesController">
```

The new part is highlighted in bold and, yes, it really is that simple.

However, pointing to the Apex controller doesn't actually load any data, or call the remote method. Like the auto-wiring between the component and (client-side) controller, this pointing simply lets these two pieces "know about" each other. This "knowing" even takes the same form, another value provider, which we'll see in a moment. But the auto-wiring only goes so far. It remains **our** job to complete the circuit.

In this case, completing the circuit means the following.

1. When our app (the expenses component) is loaded,
2. Query Salesforce for existing expense records, and
3. Add those records to the `expenses` component attribute.

We'll take each of those in turn. The first item, triggering behavior when the `expenses` component is first loaded, requires us to write an init handler. That's just a shorthand term for an action handler that's wired to a component's `init` event, which happens when the component is first created.

The wiring you need for this is a single line of markup. Add the following to the `expenses` component, right below the component's attribute definitions.

```
<aura:handler name="init" action="{!c.doInit}" value="
{!this}"/>
```

The tag is how you say that a component can, well, *handle* a specific event. In this case, we're saying that we'll handle the `init` event, and that we'll handle it with the `doInit` action handler in our controller. (Setting `value="{!this}"` marks this as a "value event." What this means is too complex to go into here. Just know that you should always add this attribute-value pair to an `init` event.)

Calling Server-Side Controller Methods

One step down, two to go. Both of the remaining steps take place in the `doInit` action handler, so let's look at it. Add the following code to the `expense` component's controller.

```
    // Load expenses from Salesforce
    doInit: function(component, event, helper) {

        // Create the action
        var action = component.get("c.getExpenses");

        // Add callback behavior for when response is received
        action.setCallback(this, function(response) {
            var state = response.getState();
            if (component.isValid() && state === "SUCCESS") {
                component.set("v.expenses",
response.getReturnValue());
            }
            else {
                console.log("Failed with state: " + state);
            }
        });

        // Send action off to be executed
        $A.enqueueAction(action);
    },
```

Before you feel lost with all the new things here, please notice that this is just another action handler. It's formatted the same, and the function signature is the same. We're in familiar territory.

That said, every line of code after the function signature is new. We'll look at all of them in a moment, but here's the outline of what this code does:

1. Create a remote method call.
2. Set up what should happen when the remote method call returns.
3. Queue up the remote method call.

That sounds pretty simple, right? Maybe the structure or specifics of the code is new, but the basic requirements of what needs to happen are once again familiar.

Does it sound like we're being really encouraging? Like, maybe we're trying to coach you through a rough patch? Well, we need to talk about something tough. The issue appears in the first line of code in the function.

```
        var action =
component.get("c.getExpenses");
```

This line of code creates our remote method call, or remote action. And at first the `component.get()` part looks familiar. We've done that many times at this point.

Except…well, before it was "v.something" that we were getting, with `v` being the value provider for the view. Here it's "c", and yes, `c` is another value provider. And we've seen a `c` value provider before, in expressions like `press="{!c.clickCreateExpense}"` and `action="{!c.doInit}"`.

Those expressions were in component markup, in the view. Here in the controller, the `c` value provider represents something different. It represents the remote Apex controller.

"Wait a minute. Are you telling me we have `c` the client-side controller, `c` the default namespace, and `c` the server-side controller, all in Lightning Components?"

Well, in a word, yes. Deep breaths.

Look, we'll be honest with you. If we had it all to do over again, we might have made some different choices. While the choices we made weren't accidents, three "c"s is definitely an opportunity for confusion. We get confused too!

But as they say, it is what it is. Forewarned is forearmed. Now you know.

Identifier	Context	Meaning
`c.`	Component markup	Client-side controller
`c.`	Controller code	Server-side controller
`c:`	Markup	Default namespace

OK, back to our code. Before we got sidetracked, we were looking at this line.

```
        var action =
component.get("c.getExpenses");
```

Where, in earlier code, `component.get("v.something")` returned to us a reference to a child component in the view (component markup), `component.get("c.whatever")` returns a reference to an action available in the controller. In this case, it returns a remote method call to our Apex controller. This is how you create a call to an `@AuraEnabled` method.

The next "line," action.setCallback(…), is a block of code that will run when the remote method call returns. Since this happens "later," let's set it aside for the moment.

The next line that actually *runs* is this one.

```
$A.enqueueAction(action);
```

We saw `$A` briefly before, but didn't discuss it. It's a framework global variable that provides a number of important functions and services. `$A.enqueueAction(action)` adds the server call that we've just configured to the Lightning Components framework request queue. It, along with other pending server requests, will be sent to the server in the next request cycle.

That sounds kind of vague. The full details are interesting, and important for advanced use of Lightning Components. But for now, here's what you need to know about `$A.enqueueAction(action)`.

- It queues up the server request.
- As far as your controller action is concerned, that's the end of it.
- You're not guaranteed when, or if, you'll hear back.

This is where that block of code we set aside comes in. But before we talk about that, a little pop culture.

Server Calls, Asynchronous Execution, and Callback Functions

Carly Rae Jepsen's single "Call Me Maybe" was released in 2011, to critical and commercial success, hitting #1 in more than a dozen countries. To date it has sold more than 18 million copies worldwide and is, apparently, one of the best-selling digital singles of all time. The most memorable line, from the chorus, is "Here's my number. So call me maybe." In addition to being upbeat and dangerously catchy, it's a metaphor for how Lightning Components handles server calls.

Hear us out. Let's look at our action handler in pseudo-code.

```
    doInit: function(component, event, helper) {
        // Load expenses from Salesforce
        var action =
component.get("c.getExpenses");
        action.setCallback(
            // Here's my number,
            // Call me maybe
        );
        $A.enqueueAction(action);
    },
```

Hmmm. Maybe we should explain the parameters to `action.setCallback()` in more detail. In the real action handler code, we call it like so.

```
        action.setCallback(this, function(response) { ...
});
```

`this` is the scope in which the callback will execute; here `this` is the action handler function itself. Think of it as an address, or...maybe a **number**. The function is what gets **called** when the server response is returned. So:

```
        action.setCallback(scope,
callbackFunction);
```

Here's my number. Call me maybe.

The overall effect is to create the request, package up the code for what to do when the request is done, and send it off to execute. At that point, the action handler itself stops running.

Here's another way to wrap your head around it. You might bundle your child up for school, and hand them a list of the chores you want them to do when they come home after classes. You drop them off at school, and then you go to work. While you're at work, you're doing *your work*, secure in the knowledge that your child, being a good kid, will do *the work you assigned to them* when they get back from school. You don't do that work yourself, and you don't know when, exactly, it will happen. But it does.

Here's one last way to look at it, again in pseudo-code. This version "unwraps" the callback function to show a more linear version of the action handler.

```
    // Not real code! Do not cut-and-paste!
    doInit: function(component, event, helper) {

        // Create server request
        var action = component.get("c.getExpenses");

        // Send server request
        $A.enqueueAction(action);

        // ... time passes ...
        // ...
        // ... Jeopardy theme plays
        // ...
        // ... at some point in the indeterminate future ...

        // Handle server response
        var state = action.response.getState();
        if (component.isValid() && state === "SUCCESS") {
            component.set("v.expenses",
action.response.getReturnValue());
        }
    },
```

We'll say it again. Asynchronous execution and callback functions are *de rigueur* for JavaScript programmers, but if you're coming from another background, it might be less familiar. Hopefully we've got it down at this point, because it's fundamental to developing apps with Lightning Components.

Handling the Server Response

Now that we've got the structure of creating a server request down, let's look at the details of how our callback function actually handles the response. Here's just the callback function.

```
    function(response) {
        var state = response.getState();
        if (component.isValid() && state === "SUCCESS") {
            component.set("v.expenses",
response.getReturnValue());
        }
    }
```

Callback functions take a single parameter, `response`, which is an opaque object that provides the returned data, if any, and various details about the status of the request.

In this specific callback function, we do the following.

1. Get the state of the response.
2. If the state is SUCCESS, that is, our request completed as planned, and if the component itself is still valid, then:
3. Set the component's `expenses` attribute to the value of the response data.

You probably have a few questions, such as:

- What happens if the response state isn't SUCCESS?
- What happens if the response never comes? (Call me *maybe*.)

- Why would the component not be valid anymore?
- How can we just assign the response data to our component attribute?

The answers to the first two are that, unfortunately, we're not going to cover those possibilities in this module. They're certainly things you need to know about and consider in your real world apps, but we just don't have the space.

Your component might not be valid if, for example, while the server is churning away, the user clicks to go to another screen. At that point, our `expenses` component might have been destroyed. In which case, trying to set a value on one of its attributes would throw an error. As the joke goes, don't do that.

The last question is the most relevant here, but it's also the easiest to answer. We defined a data type for the expenses attribute.

```
<aura:attribute name="expenses"
type="Expense__c[]"/>
```

And our server-side controller action has a method signature that defines its return data type.

```
public static List<Expense__c> getExpenses() { ...
}
```

The types match, and so we can just assign one to the other. Lightning Components handles all the details. You can certainly do your own processing of the results, and turn it into other data within your app. But if you design your server-side actions right, you don't necessarily have to.

OK, that was a lot of different ways to look at a dozen lines of code. Here's the question: Have you tried your version of our app with it yet? Because we're done with the loading expenses from Salesforce part. Go reload the app, and see if the expenses you entered in Salesforce show up!

Apex Controllers for Lightning Components

Before we take the next step in developing the app, let's dive a little deeper into that Apex controller. Here's a look at the next version, which we'll need to handle creating new records, as well as updating the Reimbursed? checkbox on existing records.

```
public with sharing class ExpensesController {

    @AuraEnabled
    public static List<Expense__c> getExpenses() {
        // Perform isAccessible() checking first, then
        return [SELECT Id, Name, Amount__c, Client__c,
Date__c,
                       Reimbursed__c, CreatedDate
                FROM Expense__c];
    }

    @AuraEnabled
    public static Expense__c saveExpense(Expense__c expense) {
        // Perform isUpdatable() checking first, then
        upsert expense;
        return expense;
    }
}
```

The earlier version promised a stern lecture, and that's coming. But first, let's focus on the details of this minimal version.

First, we've added only one new `@AuraEnabled` method, `saveExpense()`. It takes an Expense (`Expense__c`) object and upserts it. This allows us to use it to both create new records and to update existing records.

Next, notice that we created the class with the `with sharing` keywords. This automatically applies your org's sharing rules to the records that are available via these methods. For example, users would normally only see their own expense records. Salesforce handles all the complicated SOQL rules for you, automatically, behind the scenes.

Using the `with sharing` keywords is one of the essential security measures you need to take when writing server-side controller code. However, it's a measure that's necessary, but not sufficient. Do you see the comments about performing `isAccessible()` and `isUpdatable()` checks? `with sharing` only takes you so far. In particular, you need to implement object- and field-level security (which you'll frequently see abbreviated to FLS) yourself.

For example, here's a version of our `getExpenses()` method with this security minimally implemented.

```
    @AuraEnabled
    public static List<Expense__c> getExpenses() {

        // Check to make sure all fields are accessible to this user
        String[] fieldsToCheck = new String[] {
            'Id', 'Name', 'Amount__c', 'Client__c', 'Date__c',
            'Reimbursed__c', 'CreatedDate'
        };

        Map<String,Schema.SObjectField> fieldDescribeTokens =
            Schema.SObjectType.Expense__c.fields.getMap();

        for(String field : fieldsToCheck) {
            if( ! fieldDescribeTokens.get(field).getDescribe().isAccessible())
{
                throw new System.NoAccessException();
                return null;
            }
        }

        // OK, they're cool, let 'em through
        return [SELECT Id, Name, Amount__c, Client__c, Date__c,
                       Reimbursed__c, CreatedDate
                FROM Expense__c];
    }
```

It's quite an expansion from our initial one-liner, and it's still just adequate. Also, `describe` calls are expensive. If your app is calling this method frequently, you should find a way to optimize or cache your access checks per user.

Like with SLDS, we simply don't have the space to teach all of the details of secure Apex coding. Unlike with SLDS, taking responsibility for the security of the code you write is not optional. If you haven't read the secure coding practices articles in the Resources, please add them to your queue.

OK, .

Saving Data to Salesforce

Before we implement the Add Expense form for real, no cheating, let's first look at how creating a new record is a different challenge from reading existing records. With `doInit()`, we simply read some data, and then updated the user interface of the app. Straightforward, even if we did have to get Carly Rae involved in the explanation.

Creating a new record is more involved. We're going to read values from the form, create a new expense record *locally*, send that record off to be saved on the *server*, and then, when the server tells us it's saved, update the user interface, using the record returned from the *server*.

Does that make it sounds like it's going to be really complicated? Like, maybe we're going to need the Rolling Stones and a whole album of songs to help us out with the next explanation?

Let's take a look at some code, and you can decide for yourself.

First, make sure you've saved the updated version of the Apex controller, the preceding version that includes the `saveExpense()` method.

Next, to make some concepts more clear later, let's move the field validation logic into a helper function. In the `expenses` helper, add the following code, after the `createExpense()` function.

```
    validateExpenseForm: function(component) {

        // Simplistic error checking
        var validExpense = true;

        // Name must not be blank
        var nameField = component.find("expname");
        var expname = nameField.get("v.value");
        if ($A.util.isEmpty(expname)){
            validExpense = false;
            nameField.set("v.errors", [{message:"Expense name can't be
blank."}]);
        }
        else {
            nameField.set("v.errors", null);
        }

        // Amount must be set, must be a positive number
        var amtField = component.find("amount");
        var amt = amtField.get("v.value");
        if ($A.util.isEmpty(amt) || isNaN(amt) || (amt <= 0.0)){
            validExpense = false;
            amtField.set("v.errors", [{message:"Enter an expense amount."}]);
        }
        else {
            // If the amount looks good, unset any errors...
            amtField.set("v.errors", null);
        }

        return(validExpense);
    },
```

There's nothing new here, it's just a way to consolidate all of the tedious validation logic in the helper.
With the validation logic moved, you can streamline the `expenses` controller `clickCreateExpense()` action handler to look like this.

```
    clickCreateExpense: function(component, event, helper)
{
        if(helper.validateExpenseForm(component)){
            // Create the new expense
            var newExpense = component.get("v.newExpense");
            helper.createExpense(component, newExpense);
        }
    },
```

Because we put all of the details of (and all the cheating on) creating a new expense into the helper `createExpense()` function, we don't need to make any other changes in the controller. So far, so easy?

So, all we need to do is change the `createExpense()` function in the helper, to do all those complicated things we mentioned previously. Here's that code.

```
    createExpense: function(component, expense) {
        var action = component.get("c.saveExpense");
        action.setParams({
            "expense": expense
        });
        action.setCallback(this, function(response){
            var state = response.getState();
            if (component.isValid() && state ===
"SUCCESS") {
                var expenses = component.get("v.expenses");
                expenses.push(response.getReturnValue());
                component.set("v.expenses", expenses);
            }
        });
        $A.enqueueAction(action);
    },
```

Is that as complicated as you were expecting? As many lines? We hope not!

In truth, there is only one new thing in this action handler, and it's easy to understand. Let's walk through the code.

We begin by creating the action, with `component.get("c.saveExpense")` getting the new Apex controller method. Very familiar.

Next we attach a data payload to the action. This is new. We need to send the data for the new expense up to the server. But look how easy it is! You just use `action.setParams()` and provide a JSON-style object with parameter name-parameter value pairs. The one trick, and it's important, is that your parameter name must match the parameter name used in your Apex method declaration.

Next we set the callback for the request. Again, this is what will happen when the server returns a response. If you compare this callback function with our original `createExpense` helper function, it's virtually identical (minus the disgusting hack).

Just as in the prior version, we `get()` the `expenses` attribute, `push()` a value onto it, and then `set()` it. The only real difference is, instead of `push()`ing our local version of the new expense into the array, we're `push()`ing the server's response!

Why does this work? Because the server-side method upserts the (in this case new) record, which stamps an ID on it, and then returns the resulting record. Once again the server-side and client-side data types match, so we don't have to do any extra work.

And, well, that's it. No Rolling Stones needed!

Things to Watch Out For

While we've covered all of the essentials for connecting your client-side Lightning Components code with server-side Apex code, there are a couple of things that are kind of worth pointing out *before* they bite you in the you-know-where.

The first issue is case sensitivity, and this boils down to Apex and Salesforce in general are case- **in**sensitive, but JavaScript is case-**sensitive**. That is, "Name" and "name" are the same in Apex, but different in JavaScript.

This can and will lead to absolutely maddening bugs that are completely invisible to your eyes, even when they're right in front of your face. Especially if you've been working with non-Lightning Components code on Salesforce for a while, you might no longer think about the case of object and field names, methods, and so on, at all.

So here's a best practice for you: Always use the **exact** API name of every object, field, type, class, method, entity, element, elephant, or what have you. Always, everywhere, even when it doesn't matter. That way, you won't have problems. Or, at least, not *this* problem.

The other issue we'd like to draw your attention to is the nature of "required." We can't resist repeating a famous quotation: "You keep using that word. I do not think it means what you think it means."

In the code we've written so far we've seen at least two different kinds of "required." In the markup for the Add Expense form, you see the word used two ways. For example, on the expense name field.

```
        <div class="slds-form-element slds-is-required">
            <div class="slds-form-element__control">
                <ui:inputText aura:id="expname" label="Expense
Name"
                    class="slds-input"
                    labelClass="slds-form-element__label"
                    value="{!v.newExpense.Name}"
                    required="true"/>
            </div>
       </div>
```

There's an SLDS required class on the

wrapping it, and the tag has its `required` attribute set to `true`. These both illustrate only one meaning of required, which is "set the user interface of this element to indicate the field is required." In other words, this is cosmetic only. There's no protection for the quality of your data here.

Another meaning of the word "required" is illustrated in the validation logic we wrote for the same field.

```
        if ($A.util.isEmpty(expname)){
            validExpense = false;
            nameField.set("v.errors", [{message:"Expense name can't be
blank."}]);
        }
```

The word "required" is nowhere to be seen, but that's what the validation logic enforces. You must set a value for the expense name field.

And, as far as it goes, this is great. *Your* expense form won't submit a new expense with an empty name. Unless, you know, there's a bug. Or, maybe some other widget uses your same server-side controller, but doesn't do its form validation so carefully. And so on. So, this is *some* protection for your data quality, but it's not perfect.

How do you enforce, and we mean *enforce*, a data integrity rule about, in this example, expense name? You do it server-side. And not just anywhere server-side. You put the rule in the field definition, or you encode it into a trigger. Or, if you're a belt-and-suspenders kind of engineer, as all right thinking engineers are, both.

For true data integrity, when "required" means *required*, enforce it at the lowest level possible.

Resources

Remember, this module is meant for Lightning Experience. When you launch your hands-on org, switch to Lightning Experience to complete this challenge.

Connect Components with Events

Learning Objectives

After completing this unit, you'll be able to:

- Define custom events for your apps.
- Create and fire events from a component controller.
- Create action handlers to catch and handle events that other components send.
- Refactor a big component into smaller components.

Connect Components with Events

In this unit we're going to tackle the last piece of unfinished functionality in our little expenses app: the Reimbursed? checkbox. You're probably thinking that implementing a checkbox would be a short topic. We could certainly take some shortcuts, and make it a *very* short topic.

But this unit, in addition to making the checkbox work, is about removing all of the shortcuts we've already taken. We're going to spend this unit "Doing It Right." Which, in a few places, means refactoring work we did earlier.

Before we start on that, let's first talk about the shortcuts we took, the Right Way, and why the Right Way is (a little bit) harder, but also better.

Composition and Decomposition

If you take a look at our little expenses app in source code form, and list the separate code artifacts, you'll come up with something like the following.

- expenses component
  - expenses.cmp
  - expensesController.js
  - expensesHelper.js
- expensesList component
  - expensesList.cmp
- expenseItem component
  - expenseItem.cmp
- ExpensesController (server-side)
  - ExpensesController.apex

But, if you look at the app on screen, what do you see? What you should see, and what you'll eventually see everywhere you look, is that the app breaks down into many more components. You'll see that you can *decompose* our app further, into smaller pieces, than we've done so far. At the very least, we hope you see that the Add Expense form really should be its own, separate component. (That's why we drew a box around it in the user interface!)

Why didn't we make that form a separate component? Not doing that is by far the biggest shortcut we took over the course of this module. It's worse than the hack we called "disgusting," in terms of software design. The right way to build a Lightning Components app is to create independent components, and then *compose* them together to build new, higher level features. Why didn't we take that approach?

We took the shortcut, we kept the Add Expense form inside the main `expenses` component, because it kept the main `expenses` array component attribute and the controller code that affected it in the same component. We wanted the `createExpense()` helper function to be able to touch the `expenses` array directly. If we'd moved the Add Expense form into a separate component, that wouldn't have been possible.

Why not? We covered the reason briefly very early on, but we want to really hammer on it now. Lightning components are supposed to be self-contained. They are stand-alone elements that encapsulate all of their essential functionality. A component is not allowed to reach into another component, even a child component, and alter its internals.

There are two principal ways to interact with or affect another component. The first way is one we've seen and done quite a bit of already: setting attributes on the component's tag. A component's public attributes constitute one part of its API.

The second way to interact with a component is through *events*. Like attributes, components declare the events they send out, and the events they can handle. Like attributes, these public events constitute a part of the component's public API. We've actually used and handled events already, but the events have been hiding behind some convenience features. In this unit, we'll drag events out into the light, and create a few of our own.

The Wiring-a-Circuit Metaphor, Yet Again

These two mechanisms—attributes and events—are the API "sockets," the ways you connect components together to form complete circuits. Events are also, behind the scenes, the electrons flowing through that circuit. But that's only one way that events are different from attributes.

When you set the `press` attribute on a to an action handler in a component's controller, you create a direct relationship between those two components. They are linked, and while they're using public APIs to remain independent of each other, they're still coupled.

Events are different. Components don't send events *to* another component. That's not how events work. Components broadcast events of a particular type. If there's a component that responds to that type of event, and if that component "hears" your event, then it will act on it.

You can think of the difference between attributes and events as the difference between wired circuits and wire**less** circuits. And we're not talking wireless phones here. One component doesn't get the "number" for another component and call it up. That would be an attribute. No, events are like wireless broadcasts. Your component gets on the radio, and sends out a message. Is there anyone out there with their radio set turned on, and tuned to the right frequency? Your component has no way of knowing—so you should write your components in such a way that it's OK if no one hears the events they broadcast. (That is, things might not work, but nothing should crash.)

Sending an Event from a Component

OK, enough theory, let's do something specific with our app, and see how events work in code. We will start by implementing the Reimbursed? checkbox. Then we'll take what we learned doing that, and use it to refactor the Add Expense form into its own component, the way the Great Engineer intended.

Let's start with the `expenseItem` component. You created this component earlier, but this version includes SLDS markup to make it a little more stylish. It also includes conditional styling to make unreimbursed expenses more noticeable. Replace your version with the following markup.

```
<aura:component>
    <aura:attribute name="expense" type="Expense__c"/>

    <div class="slds-card">

        <!-- Color the item green if the expense is reimbursed -->
        <div class="{!v.expense.Reimbursed__c == true ?
            'slds-theme--success' : 'slds-theme--warning'}">

            <header class="slds-card__header slds-grid grid--flex-spread">
                <a aura:id="expense" href="{!'/' + v.expense.Id}">
                    <h3>{!v.expense.Name}</h3>
                </a>
            </header>

            <section class="slds-card__body">
                <div class="slds-tile slds-hint-parent">
                    <p class="slds-tile__title slds-truncate">Amount:
                        <ui:outputCurrency value="
{!v.expense.Amount__c}"/>
                    </p>
                    <p class="slds-truncate">Client:
                        <ui:outputText value="{!v.expense.Client__c}"/>
                    </p>
                    <p class="slds-truncate">Date:
                        <ui:outputDate value="{!v.expense.Date__c}"/>
                    </p>
                    <p class="slds-truncate">Reimbursed?
                        <ui:inputCheckbox value="
{!v.expense.Reimbursed__c}"
                                  click="{!c.clickReimbursed}"/>
                    </p>
                </div>
            </section>
        </div>
    </div>

</aura:component>
```

Another new thing we have here is a click handler on the for the `Reimbursed__c` field. The `click` attribute of is similar to the `press` attribute of . It gives us an easy way to wire the checkbox to an action handler that updates the record when it's checked or unchecked.

Before we look at an implementation of the `clickReimbursed` handler, let's have a think about what should happen when it's checked or unchecked. Based on the code we wrote to create a new expense, updating an expense is probably something like this.

1. Get the `expense` item that changed.
2. Create a server action to update the event.
3. Package `expense` into the action.
4. Set up a callback to handle the response.
5. Fire the action, sending the request to the server.
6. When the response comes and the callback runs, update the `expenses` attribute.

Um, what `expenses` attribute? Look at our component markup again. No `expenses`, just a singular `expense`. Hmmm, right, this component is just for a single item. There's an `expenses` attribute on the `expensesList` component…but that's not even the "real" `expenses`. The real one is a component attribute in the top-level `expenses` component. Hmmm.

Is there a `component.get("v.parent")`? Or would it have to be `component.get("v.parent").get("v.parent")`—something that would let us get a reference to our parent's parent, so we can set `expenses` there?

Stop. Right. There.

Components do not reach into other components and set values on them. There's no way to say "Hey grandparent, I'm gonna update `expenses`." Components keep their hands to themselves. When a component wants an ancestor component to make a change to something, it **asks**. Nicely. By sending an event.

Here's the cool part. Sending an event looks almost the same as handling the update directly. Here's the code for the `clickReimbursed` handler.

```
({
    clickReimbursed: function(component, event, helper) {
        var expense = component.get("v.expense");
        var updateEvent =
component.getEvent("updateExpense");
        updateEvent.setParams({ "expense": expense });
        updateEvent.fire();
    }
})
```

Whoa. That's pretty simple! And it does look kind of like what we envisioned above. The preceding code for `clickReimbursed` does the following:

1. Gets the `expense` that changed.
2. Creates an event named `updateExpense`.
3. Packages `expense` into the event.
4. Fires the event.

The callback stuff is missing, but otherwise this is familiar. But…what's going to handle calling the server, and the server response, and update the main `expenses` array attribute? And how do we know about this `updateExpense` event, anyway?

`updateExpense` is a custom event, that is, an event we write ourselves. You can tell because, unlike getting a server action, we use `component.getEvent()` instead of `component.get()`. Also, what we are getting doesn't have a value provider, just a name. We'll define this event in just a moment.

As for what's going to handle calling the server and handling the response, let's talk about it. We could implement the server request and handle the response right here in the `expenseItem` component. Then we'd send an event to just rerender things that depend on the `expenses` array. That would be a perfectly valid design choice, and would keep the `expenseItem` component totally self-contained, which is desirable.

However, as we'll see, the code for creating a new expense and the code for updating an existing expense are very similar, enough so that we'd prefer to avoid duplicate code. So, the design choice we've made is to send an `updateExpense` event, which the main `expenses` component will handle. Later, when we refactor our form, we'll do the same for creating a new expense.

By having all child components delegate responsibility for handling server requests and for managing the `expenses` array attribute, we're breaking encapsulation a bit. But, if you think of these child components as the internal implementation details of the `expenses` component, that's OK. The main `expenses` component *is* self-contained.

You have a choice: consolidation of critical logic, or encapsulation. You'll make trade-offs in Lightning Components just like you make trade-offs in any software design. Just make sure you document the details.

Defining an Event

The first thing we'll do is define our custom event. In the Developer Console, select , and name the event "expensesItemUpdate". Replace the default contents with the following markup.

```
<aura:event type="COMPONENT">
    <aura:attribute name="expense"
type="Expense__c"/>
</aura:event>
```

There are two types of events, component and application. Here we're using a component event, because we want an ancestor component to catch and handle the event. An ancestor is a component "above" this one in the component hierarchy. If we wanted a "general broadcast" kind of event, where *any* component could receive it, we'd use an application event instead.

The full differences and correct usage of application vs. component events isn't something we're able to get into here. It's a more advanced topic, and there are enough complicated details that it's a distraction from our purpose in this module. When you're ready for more, the Resources will help you out.

The other thing to notice about the event is how compact the definition is. We named the event when it was created, `expensesItemUpdate`, and its markup is a beginning and ending tag, and one tag. An event's attributes describe the payload it can carry. In the `clickReimbursed` action handler, we set the payload with a call to `setParams()`. Here in the event definition, we see how the event parameter is defined, and that there are no other valid parameters.

And that's pretty much all there is to defining events. You don't add implementation or behavior details to events themselves. They're just packages. In fact, some events don't have any parameters at all. They're just messages. "This happened!" All of the behavior about what to do if "this" happens is defined in the components that send and receive the event.

Sending an Event

We already looked at how to actually *fire* an event, in the `clickReimbursed` action handler. But for that to work, we need to do one last thing, and that's register the event. Add this line of markup to the `expenseItem` component, right below its attribute definitions.

```
    <aura:registerEvent name="updateExpense"
type="c:expensesItemUpdate"/>
```

This markup says that our component fires an event, named "updateExpense", of type "c:expensesItemUpdate". But, wasn't "expensesItemUpdate" the *name* of the event when we defined it? And what happened to component or application event *types*?

You're right to think it's a little confusing—it really is a bit of a switch-a-roo. It might help to think of "application" and "component" as Lightning Components *framework* event types, while the types that come from the names of events you define are custom event types, or event *structure* types. That is, when you define an event, you define a package format. When you register to send an event, you declare what format it uses.

The process of defining and registering an event might still seem a bit weird, so let's look ahead a bit. Here in `expenseItem`, we're going to send an event named `updateExpense`. Later in `expenseForm`, we're going to send an event named `createExpense`. Both of these events need to include an expense to be saved to the server. And so they both use the `c:expensesItemUpdate` event type, or package format, to send their events.

On the receiving end, our main `expenses` component is going to register to handle *both* of these events. Although the server call ends up being the same, the user interface updates are slightly different. So how does `expenses` know whether to create or update the expense in the `c:expensesItemUpdate` package? By the *name* of the event being sent.

Understanding the distinction here, and how one event can be used for multiple purposes, is a light bulb moment in learning Lightning Components. If you haven't had that moment quite yet, you'll have it when you look at the rest of the code.

Before we move on to handling events, let's summarize what it takes to send them.

1. Define a custom event by creating a Lightning Event, giving it a name and attributes.
2. Register your component to send these events, by choosing a custom event type and giving this specific use of that type a name.
3. Fire the event in your controller (or helper) code by:
    a. Using `component.getEvent()` to create a specific event instance.
    b. Sending the event with `fire()`.

If you went ahead and implemented all of the code we just looked at, you can test it out. Reload your app, and toggle a Reimbursed? checkbox a few times. If you missed a step, you'll get an error, and you should recheck your work. If you did everything right…hey, wait, the expense changes color to show its Reimbursed? status, just as expected!

This behavior was present before we even started this unit. That's the effect of the component having `value="{!v.expense.Reimbursed__c}"` set. When you toggle the checkbox, the *local* version of the `expense` is updated. But that change isn't being sent up to the server. If you look at the expense record in Salesforce, or reload the app, you won't see the change.

Why not? We've only done *half* of the work to create a complete circuit for our event. We have to finish wiring the circuit by creating the event handler on the other side. That event handler will take care of sending the change to the server, and making the update durable.

Handling an Event

Enabling the `expenseItem` component to send an event required three steps. Enabling the `expenses` component to receive and handle these events requires three parallel steps.

1. Define a custom event. We've already done this, because `expenseItem` is sending the same custom event that `expenses` is receiving.
2. Register the component to handle the event. This maps the event to an action handler.
3. Actually handle the event in an action handler.

Since we've already done step 1, let's immediately turn to step 2, and register `expenses` to receive and handle the `updateExpense` event. Like registering to send an event, registering to handle one is a single line of markup, which you should add to the `expenses` component right after the init handler.

```
     <aura:handler name="updateExpense"
event="c:expensesItemUpdate"
        action="{!c.handleUpdateExpense}"/>
```

Like the init handler, this uses the tag, and has an `action` attribute that sets the controller action handler for the event. Like when you registered the event in `expenseItem`, here you set the name and type of the event—though note the use of the much more sensibly named `event` attribute for the type.

In other words, there's not much here you haven't seen before. What's new, and specific to handling custom events, is the combination of the attributes,

and knowing how this receiver “socket” in `expenses` matches up with the sender “socket” in `expenseItem`.

This completes the wiring part of making this work. All that’s left is to actually write the action handler!

We’ll start with the `handleUpdateExpense` action handler. Here’s the code, and be sure to put it riiiiight under the `handleCreateExpense` action handler.

```
    handleUpdateExpense: function(component, event, helper)
{
        var updatedExp = event.getParam("expense");
        helper.updateExpense(component, updatedExp);
    }
```

Huh. That’s interesting. Except for the form validation check and the specific helper function we’re delegating the work to, it looks like this action handler is the same as `handleCreateExpense`.

And now let’s add the `updateExpense` helper function. As we did with the action handler, make sure you put this code right below the `createExpense` helper function.

```
    updateExpense: function(component, expense) {
        var action = component.get("c.saveExpense");
        action.setParams({
            "expense": expense
        });
        action.setCallback(this, function(response){
            var state = response.getState();
            if (component.isValid() && state ===
"SUCCESS") {
                // do nothing!
            }
        });
        $A.enqueueAction(action);
    },
```

Two things you should notice right off the bat. First, except for the callback specifics, the `updateExpense` helper method is identical to the `createExpense` helper method. That smells like opportunity.

Second, about those callback specifics. What gives? How can the right thing to do be *nothing*?

Think about it for a moment. Earlier, when testing sending the event (if not before), we saw that the `expenseItem` component’s color changed in response to toggling the Reimbursed? checkbox. Remember the explanation? The local copy of the expense record is *already updated!* So, at least for the moment, when the server tells us it was successful at updating its version, we don’t have to do anything.

Note that this code only handles the case where the server is *successful* at updating the expense record. We’d definitely have some work to do if there was an error. Say, Accounting flagged this expense as non-reimbursable, making it impossible to set this field to `true`. But that, as they say, is a lesson for another day.

Refactor the Helper Functions

Let’s go back to that opportunity we saw to factor out some common code. The two helper functions are identical except for the callback. So, let’s make a new, more generalized function that takes the callback as a parameter.

```
    saveExpense: function(component, expense, callback)
{
        var action = component.get("c.saveExpense");
        action.setParams({
            "expense": expense
        });
        if (callback) {
            action.setCallback(this, callback);
        }
        $A.enqueueAction(action);
    },
```

The `callback` parameter is optional. If it’s there, we’ll pass it along to `action`. Simple. And now we can reduce our event-specific helper functions to the following code.

```
    createExpense: function(component, expense) {
        this.saveExpense(component, expense, function(response)
{
            var state = response.getState();
            if (component.isValid() && state === "SUCCESS") {
                var expenses = component.get("v.expenses");
                expenses.push(response.getReturnValue());
                component.set("v.expenses", expenses);
            }
        });
    },

    updateExpense: function(component, expense) {
        this.saveExpense(component, expense);
    },
```

`createExpense` is only a little shorter, but it's exclusively focused on what to do when the response comes back (the callback). And wow, `updateExpense` is a one-liner!

Refactor the Add Expense Form

That little refactoring exercise was so satisfying, and using events was so (we're sorry) electrifying, let's do it again, but bigger. *More cowbell!*

This next task involves extracting the Add Expense form from the expenses component, and moving it to its own, new component. Extracting the form markup is easy enough, a simple copy-and-paste exercise. But what else moves with it? Before we start moving pieces around willy-nilly, let's think about what moves and what stays.

In the current design, the form's action handler, `clickCreateExpense`, handles input validation, sending the request to the server, and updating local state and user interface elements. The form will still need an action handler, and should probably still handle form validation. But we'll plan on having the rest stay behind, because we're keeping our server request logic consolidated in the `expenses` component.

So there's a little (but only a little!) teasing apart to do there. Our plan, then, is to start by moving the form markup, and then move as little as possible after that to make it work correctly. We'll refactor both components to communicate via events, instead of via direct access to the `expenses` array component attribute.

Let's get started!

In the main `expenses` component, select everything between the two comments, including the beginning and ending comments themselves. Cut it to your clipboard. (Yes, *cut*. We're committed.)

Create a new Lightning component, and name it "expenseForm". Paste the copied Add Expense form markup into the new component, between the tags.

Back to the `expenses` component. Add the new `expenseForm` component to the markup. That section of `expenses` should look like this.

```
    <!-- NEW EXPENSE FORM -->
    <div class="slds-col slds-col--padded slds-p-top--
large">

        <c:expenseForm/>

    </div>
    <!-- / NEW EXPENSE FORM -->
```

At this point, you can reload your app to see the changes. There should be no *visible* changes. But, unsurprisingly, the Create Expense button no longer works.

Let's get quickly through the rest of the moving things around part.

Next, move the `newExpense` attribute from the `expenses` component to the `expenseForm` component markup. This is used for the form fields, so it needs to be in the form component. It moves over with no changes required, so just cut from one and paste in the other.

In the `expenseForm` component, create the controller and helper resources.

Move the `clickCreateExpense` action handler from the `expenses` controller to the `expenseForm` controller. The button is in the form component, and so the action handler for the button needs to be there, too. Believe it or not, this also needs no changes. (You might begin sensing a theme here.)

Move the `validateExpenseForm` function from the `expenses` helper to the `expenseForm` helper. Again, no changes required.

Now we need to make a couple of actual changes. But these will be familiar, because we're just adding event sending, which we did before for

`expenseItem`. `expenseItem`, you'll recall, also sends an event with an `expense` payload, which is handled by the `expenses` component.

In the `expenseForm` helper, create the `createExpense` function.

```
    createExpense: function(component, newExpense) {
        var createEvent =
component.getEvent("createExpense");
        createEvent.setParams({ "expense": newExpense });
        createEvent.fire();
    },
```

This looks very much like the `clickReimbursed` action handler in `expenseItem`.

If a component is going to send an event, it needs to register the event. Add the following to the `expenseForm` component markup, just below the `newExpense` attribute.

```
    <aura:registerEvent name="createExpense"
type="c:expensesItemUpdate"/>
```

At this point, we've done all the work to implement the `expenseForm` component. You should be able to reload the app, and the form now "works" in the sense that there are no errors, and you should see the appropriate form messages when you enter invalid data. If you're using the Salesforce Lightning Inspector, you can even see that the `expensesItemUpdate` event is being fired. All that's left is to handle it.

Before we handle the event, please do notice how easy this refactoring was. Most of the code didn't change. There's a total of six lines of new code and markup in the preceding steps. It's unfamiliar to do this work today, but do it a few times, and you realize that you're just moving a bit of code around.

OK, let's finish this. The `expenseForm` fires the `createExpense` event, but we also need the `expenses` component to catch it. First we register the `createExpense` event handler, and wire it to the `handleCreateExpense` action handler. Once again, this is a single line of markup. Add this line right above or below the `updateExpense` event handler.

```
    <aura:handler name="createExpense"
event="c:expensesItemUpdate"
        action="{!c.handleCreateExpense}"/>
```

Finally, for the last step, create the `handleCreateExpense` action handler in the `expenses` controller. Add this code right above or below the handleUpdateExpense action handler.

```
    handleCreateExpense: function(component, event, helper)
{
        var newExpense = event.getParam("expense");
        helper.createExpense(component, newExpense);
    },
```

Yep, that simple. All of the work is delegated to the `createExpense` helper function, and that didn't move or change. Our `handleCreateExpense` action handler is just there to wire the right things together.

And with that, we're finished showing how to loosely couple components using events. Create and fire the event in one component, catch and handle it in another. Wireless circuits!

Bonus Lesson—Minor Visual Improvements

Before we head off into the sunset, or rather, the challenge, here are two modest visual improvements.

First, have you noticed that the little calendar icon for our form field is off in left field?

No joke, the placement of that icon has been driving your humble author nuts this entire module. If it's been bothering you, too, here's how to fix it. (We held off on having you do this until we'd refactored the form into its own component.) In the `expenseForm` component, add a style resource by clicking the **STYLE** button. Replace the default style rules with the following.

```
.THIS .uiInputDate .datePicker-openIcon
{
    position: absolute;
    left: 95%;
    top: 55%;
}
```

Reload the app for sweet, sweet relief.

Last, we'd like to improve the layout of our app a little bit, by adding some more SLDS markup. This last bit also gives you an opportunity to see the full `expense` component after all our changes. In the `expense` component, replace all of the markup with the following.

```
<aura:component controller="ExpensesController">

    <aura:attribute name="expenses" type="Expense__c[]"/>

    <aura:handler name="init" action="{!c.doInit}" value="{!this}"/>
    <aura:handler name="createExpense" event="c:expensesItemUpdate"
        action="{!c.handleCreateExpense}"/>
    <aura:handler name="updateExpense" event="c:expensesItemUpdate"
        action="{!c.handleUpdateExpense}"/>

    <!-- PAGE HEADER -->
    <div class="slds-page-header" role="banner">
      <div class="slds-grid">
        <div class="slds-col">
          <p class="slds-text-heading--label">Expenses</p>
          <h1 class="slds-text-heading--medium">My Expenses</h1>
        </div>
      </div>
    </div>
    <!-- / PAGE HEADER -->

    <!-- NEW EXPENSE FORM -->
    <div class="slds-col slds-col--padded slds-p-top--large">

        <c:expenseForm/>

    </div>
    <!-- / NEW EXPENSE FORM -->

    <!-- EXISTING EXPENSES -->
    <div class="slds-grid slds-m-top--large">

        <!-- EXPENSES LIST -->
        <div class="slds-col slds-col-rule--right slds-p-around--
small
            slds-size--8-of-12">
            <c:expensesList expenses="{!v.expenses}"/>
        </div>
        <!-- / EXPENSES LIST -->

        <!-- SOMETHING COOL -->
        <div class="slds-col slds-p-left--large slds-size--4-of-12">
            <!-- Bonus lesson, coming soon.
                 Watch this space for details. -->
        </div>
        <!-- / SOMETHING COOL -->

    </div>
    <!-- / EXISTING EXPENSES -->

</aura:component>
```

The effects of the changes are to add some margins and padding, and make the expenses list more narrow. The layout leaves room to put something over there on the right. In the next unit, we'll suggest a couple of exercises you could do on your own.

Resources

Remember, this module is meant for Lightning Experience. When you launch your hands-on org, switch to Lightning Experience to complete this challenge.

Discover Next Steps

Learning Objectives

After completing this unit, you'll be able to:

- List five aspects of Lightning Components that you can explore in more depth.
- Plan three improvements you could make to the Expenses app.
- Earn. This. Badge!

Congratulations!

You made it! We are thrilled for you, impressed by you, and proud of you. Seriously, this is a hard module, and earning this badge says a lot. Congratulations!

This unit's challenge is all that stands between you and that badge, and after all the effort you've put in so far, we're making this one easy. But, please don't skip ahead! While this unit is "easy" in the challenge sense, if you want to be successful as a Lightning Components developer, this unit is every bit as important as all of the others.

The Lightning Components Basics module teaches the fundamentals of developing apps with Lightning Components, but it's just the start. As long as this module has been to this point, there's actually more to learn still ahead of you. At least, if you want to become a Lightning Components master.

In this unit we'll cover two topics. First, we'll provide a survey of the kinds of things we didn't have space to cover in this module, along with pointers for where you can start to learn about them. And after that we'll suggest a couple of homework-style projects you could take on. There's a number of "obvious" additions to our little expenses app that you could try to do yourself. The best way to learn is by doing!

What We Skimmed Over or Zipped Past

As we were building the expenses app we had to say "we're not going to cover this here" more times than we can count. We hated every one of them, and you probably did, too. But we believed you would hate us more if we filled up this module to eight hours or more.

Salesforce Lightning Design System

The SLDS is a robust, flexible, and comprehensive system for implementing the Lighting Experience style in your apps. You can use it in Lightning Components, Visualforce, and even in plain markup. A writer on the Salesforce documentation team used it to create prototypes of…wait, we can't tell you about that yet.

SLDS is fun to learn, and even more fun to use. There's a Trailhead module dedicated to it (using Visualforce), and number of Trailhead projects that illustrate its use. And there's a *wowza* web site dedicated to getting it into people's hands.

Where You Can Use Lightning Components

We blazed through this topic in a flurry of screenshots, and we never looked back. But Lightning Components can be used in many, many ways within Salesforce, and even outside of it.

In particular, we spent all of our time building a stand-alone "my.app". You'll definitely want to learn how to add your apps to Salesforce1 and Lighting Experience. The good news is that it's so easy, it's going to make you wonder why we didn't cover it. (Read on, partner.)

The Lightning Components Developer Guide is the best source of information about where and how you can use your Lightning Components apps.

Debugging

We covered only the most primitive debugging techniques. Learning to debug your Lightning Components apps, using several different sophisticated tools for doing so, is something that will pay dividends, in the form of hours of your life back, and hair not pulled out.

In particular, we recommend learning Chrome's rich DevTools suite, and the Salesforce Lightning Inspector that works within it. Apex developers will also want to learn about the debugging tools available for Apex, many of them directly in the Developer Console.

Data Types

We briefly mentioned that there are some "framework-specific" data types you can use for attributes. Chiefly these are used for facets, in particular the `body` facet. And we deliberately skipped facets, especially `body`, because they're complicated, and they're not essential for basic Lightning Components development. You'll want to learn all about these concepts eventually, though, because being able to set a component's `body` (or other facets) is a

powerful technique that can save you a lot of code and awkwardness.

Helpers

You wrote a fair bit of helper code in this module, but there are some advanced uses of helpers that are worth knowing. Since helpers are the chief way you share reusable code, it's an important area to explore.

- Sharing JavaScript Code in a Component Bundle
- Using JavaScript

Server Request Lifecycle and Handling

We covered how to make a server request using `$A.enqueueAction()` and how to handle a successful response. There are other possibilities, and you really need to know how to handle them. There are also many different *kinds* of requests, and using the right one can significantly improve your app's performance in some cases.

Security, Security, Security, Security

We lectured earlier, so we won't here. But there's a lot to know.

Application Events

We mentioned these, and they're important for larger apps. And while we covered event basics, there's a lot to learn about them. You can't build sophisticated apps with Lightning Components without learning all about events.

What We Didn't Cover At All

There's also a number of topics we simply didn't mention. Some of these are complex, some are advanced, and some are both. We're listing a few here for your consideration.

Other Bundle Resource Types

We covered the four core Lightning Components bundle resource types. There are four others. While the Design and SVG resources have somewhat specialized purposes, the Documentation and Renderer resources are potentially useful in any Lightning component.

Navigation

You would be forgiven for thinking this is fundamental. In any complex, real-world app it is. And in many cases, if you're running inside of Lightning Experience or Salesforce1, developing navigation is actually pretty easy. But, see the next item.

- URL-Centric Navigation
- force:navigateToURL

Dynamically Creating Components

One of the principal ways of "navigating" within your Lightning Components apps is to create new components dynamically, in response to user actions. This whole area is rich and powerful, and complex enough that you'd be slogging away at this module for a week. We'll give you a search term and you can start to explore: `$A.createComponent()`.

- Dynamically Creating Components

Error Handling

There's handling server errors, and there's handling errors that are purely client-side. For the purposes of this module, we assumed you'd always be successful. Sadly, in the real world, bugs and *weird* both happen with alarming reliability. The Great Engineer said it best: Anything that *can* go wrong, *will* go wrong. So you might as well plan to handle and recover from errors as best you can.

`force:` Namespace Components and Events

When your Lightning Components app runs inside Lightning Experience and Salesforce1, there are a number of really cool components and events you can use. Some of these components work in other contexts, but many of them are only easy-to-use in Lightning Experience and Salesforce1, and so we didn't look at them.

- Component Reference (see the `force:` items)
- Event Reference (see the `force:` items)

System Events

Technically we touched on this, in the form of init handlers, but we didn't explain that the init event is one of a suite of system events that you can catch during the component or app lifecycle. (For that matter, we didn't talk about that lifecycle, either.) There's a bunch of them, and you can use them to have your components "do something" at specific points in their existence, or to handle specific situations.

Exercises for the Adventurous

We hope we've whet your appetite for more Lightning Components learning. Here's a couple of ideas we have for things you could do with the Expenses app that would make for interesting exploration, and that fit naturally with what you just learned.

Clear the Form After Submission

Right now when you click the Create Expense button, the form stays filled in. It's not hard to set the fields to empty strings…but when should you do it? Think about the behavior you want, about usability, and about the various possibilities in terms of server responses. (That is, besides SUCCESS.)

Once you decide on behavior, where do you put the code? It sounds easy, at first, and then you realize the server response handling is in `expenses`, and the form fields are in `expenseForm`. What does this sound like a job for?

Display an "Expense Saved" Message

When your expense is successfully saved to the server, it would be nice to show the user a little success message. And maybe the message is different for updating the Reimbursed? checkbox vs. creating a new expense. Or, maybe sometimes you don't show a message at all.

Handle Failure

What happens if a data validation rule on a field prevents you from saving a record on the server? Or some other error happens? Right now, nothing; the app silently fails. For the Reimbursed? checkbox, the app can show an incorrect state for the expense. Both of these are bad!

Dealing with simple failures isn't actually that many lines of code. But you'll need to do some reading before you start.

Allow Expense Record Editing

This one is kind of advanced, but you can tackle it in stages. First, when you click on an expense in the expense list, have the form fill in with the appropriate expense values. Then have it change the text of the Create Expense button to Save Expense. (Don't cheat and have it say Save Expense all the time!) Then change the event that the form fires to update the existing expense instead of creating a new one.

And, with that, we'll leave you for today. Congratulations again, and we hope you'll find fame, fortune, and excitement in your Lightning Components app development adventure!

Introduction to Salesforce Connect

Learning Objectives

After completing this unit, you'll be able to:

- Explain what Salesforce Connect is.
- Describe two typical use cases for Salesforce Connect.
- Explain how Salesforce Connect differs from extract, transform, and load (ETL) tools.
- Explain how external objects differ from standard and custom objects.

Overview of Salesforce Connect

Salesforce Connect is a framework that enables you to view, search, and modify data that's stored outside your Salesforce org. For example, perhaps you have data that's stored on premises in an enterprise resource planning (ERP) system. Instead of copying the data into your org, you can use external objects to access the data in real time via web service callouts.

Salesforce Connect lets your Salesforce org access data from a wide variety of external systems. You can integrate tables from SAP® NetWeaver Gateway, Microsoft Dynamics® NAV, and many other data sources in real time without writing a single line of code. Previously, the only way to integrate external data with Salesforce was to use extract, transform, and load (ETL) tools. That process is time consuming and requires you to copy data into your org that you might never use or quickly becomes stale. In contrast, Salesforce Connect maps data tables in external systems to external objects in your org.

External objects are similar to custom objects, except that they map to data located outside your Salesforce org. External object data is always up to date. Salesforce Connect provides a live connection to external data rather than a copy that consumes storage and must be regularly synced. Accessing an external object fetches the data from the external system in real time.

We recommend that you use Salesforce Connect if most of these conditions apply.

- You have a large amount of data that you don't want to copy into your Salesforce org.
- You need small amounts of data at any one time.

- You need real-time access to the latest data.
- You store your data in the cloud or in a back-office system, but want to display or process that data in your Salesforce org.

External Objects vs. Custom Objects

External objects share much of the same functionality as custom objects. For example, you can:

- Access external objects via list views, detail pages, record feeds, custom tabs, and page layouts.
- Define relationships between external objects and standard or custom objects to integrate data from different sources.
- Enable Chatter feeds on external object pages for collaboration.
- Enable create, edit, and delete operations on external objects.

If you need frequent access to large amounts of external data, ETL might still be your best option for optimal performance. External objects are not a replacement for ETL. They are a complementary approach for accessing external data that provide great benefits, including seamless integration with the Salesforce Platform, including our APIs, mobile, Chatter, and more. For example, external objects are available to standard Salesforce tools such as the Salesforce1 mobile app, global search, SOSL and SOQL queries, Apex, Visualforce, APIs, change sets, and packages.

Here is a quick comparison of the features supported in external objects and custom objects.

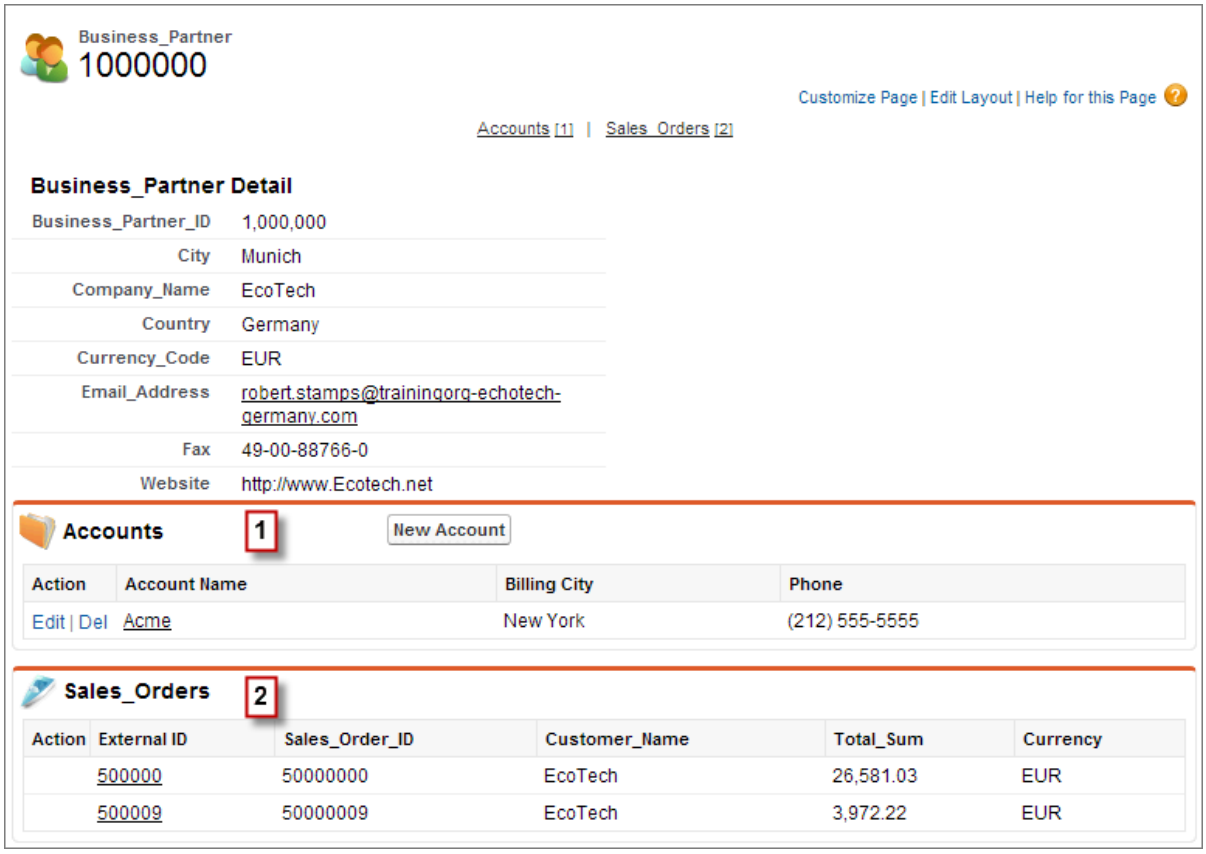
Feature	Custom Objects	External Objects
Data is stored in your Salesforce org	Yes	No
Read	Yes	Yes
Write	Yes	Yes (limited)
Tabs, layouts	Yes	Yes
Visualforce	Yes	Yes
Field-level security	Yes	Yes
Sharing	Yes	No
REST and SOAP API	Yes	Yes
SOQL	Yes	Yes (limited)
Search and SOSL	Yes	Yes (pass-through)
Formula fields	Yes	Not Yet
Workflow, triggers	Yes	Not Yet
Reports and analytics	Yes	Yes (limited)
Chatter	Yes	Yes (no field tracking)

A Salesforce Connect Example

Suppose you have product order information stored in an external database, and you want to view those orders as a related list on each account record in Salesforce. Salesforce Connect enables you to set up a relationship between the parent account object and the child external object for orders. Then you can set up the page layouts for the parent object to include a related list that displays child records.

The figure below shows how Salesforce Connect can provide a seamless view of data across system boundaries. A record detail page for the Business_Partner external object includes two related lists of child objects.

- Account standard object (1)
- Sales_Order external object (2)



In this example, external lookup relationships and page layouts enable users to view related data that's stored both inside and outside the Salesforce org on a single page.

Types of External Connections

To connect to data that's stored on an external system, Salesforce Connect uses one of these specially designed adapters.

- **OData 2.0 adapter** or **OData 4.0 adapter**—Connects to data exposed by any OData 2.0 or 4.0 producer on the Internet. OData (Open Data Protocol) is a modern, REST-based protocol for integrating data. Vendors such as SAP and Microsoft have already implemented OData support, so products such as NetWeaver and SharePoint are directly accessible. Integration products from Salesforce partners extend the reach of Salesforce Connect to a much wider range of back-office systems.
- **Cross-org adapter**—Connects to data that's stored in another Salesforce org. The Cross-org adapter uses the standard Force.com REST API. It directly connects to the other org without the need of an intermediary web service, as is the case with OData.
- **Custom adapter created via Apex**—If the OData and cross-org adapters aren't suitable for your needs, develop your own adapter with the Apex Connector Framework.

Resources

Set up Salesforce Connect

Learning Objectives

After completing this unit, you'll be able to:

- Create an external data source in your Salesforce org to specify how to connect to an external system.
- Validate the connection to the external system.
- Sync to create external objects that map to the external system's schema.
- View the external objects.

Overview of Setup

Setting up external data integration with Salesforce Connect involves these high-level steps.

1. **Create the external data source.** If your external system hosts multiple services, create an external data source for each service that's required to access the data.

2. **Create the external objects and their fields.** Create an external object in your Salesforce org for each external data table that you want to access. On each external object, create a custom field for each external table column that you want to access from your Salesforce org.

   Note

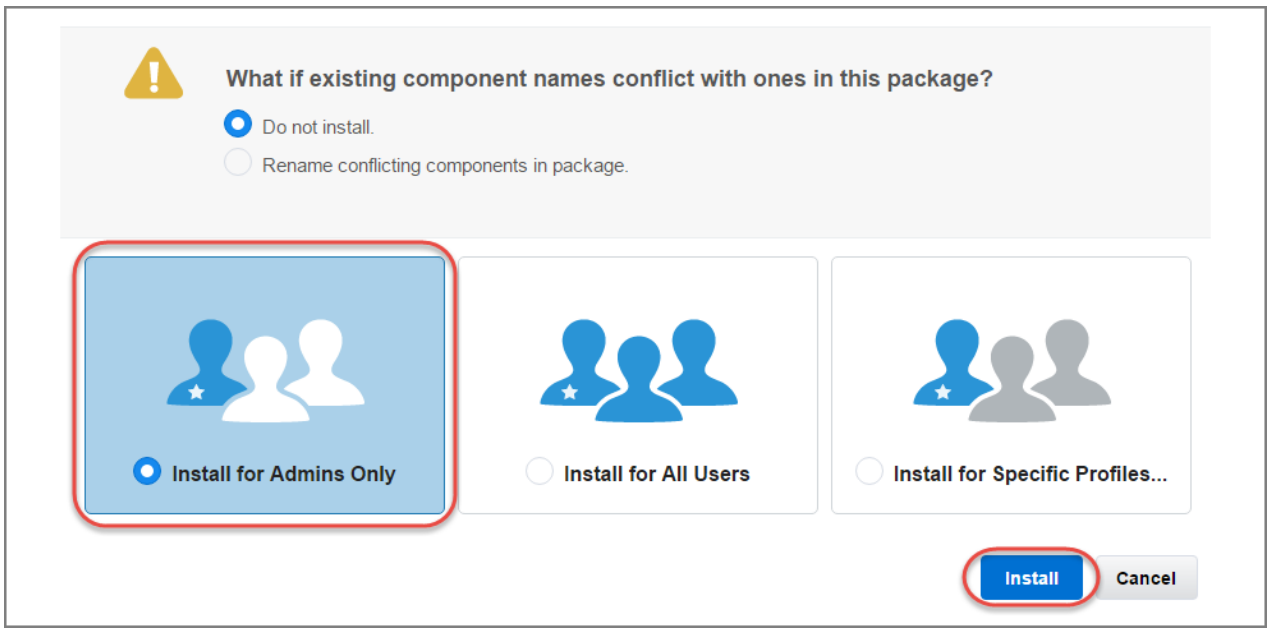
   If the external system allows it, we recommend that you sync the external data source to automatically create related external objects. You can instead choose to manually define external objects to customize the external object names and manually create the custom fields.

3. **Define relationships for the external objects.** Create lookup, external lookup, and indirect lookup relationship fields to provide seamless views of data across system boundaries.
4. **Enable user access to external objects and their fields.** Grant object and field permissions through permissions sets or profiles.
5. **Set up user authentication.** For each external data source that uses per-user authentication, do both of the following.
   a. **Enable users to authenticate to the external data source.** Grant users access through permission sets or profiles.
   b. **Set up each user's authentication settings.** Tell your users how to set up and manage their own authentication settings for external systems in their personal settings. Alternatively, you can perform this task for each user.

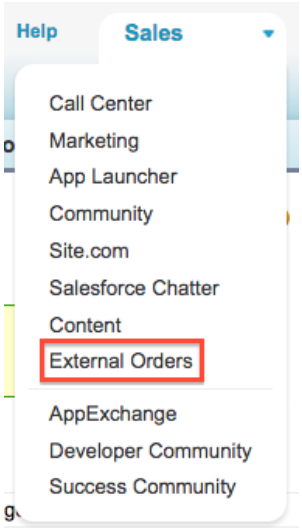
As part of this module, you integrate sample order data with the existing account data in your Salesforce Developer Edition. To go through the steps, you need to install a test package that configures the necessary schema on the account object, creates a Customer ID field, and assigns a value for Customer ID to each account.

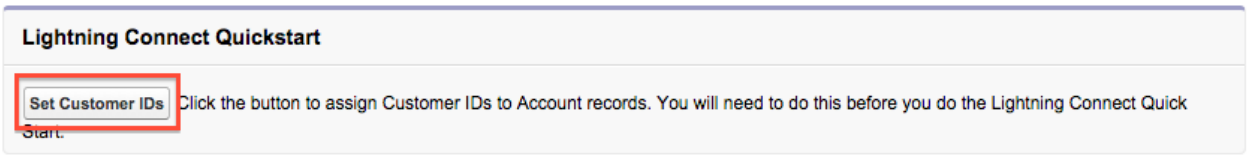
Follow these steps to install the package.

1. Open a browser, and click here to start the installation.
2. Select **Install for Admins Only**.
3. Click **Install**.



4. Click the app menu (top right), then select **External Orders**.
5. Click **Set Customer IDs** to assign customer ID numbers to the sample account records in your Developer Edition.





Your Salesforce Developer Edition is now set up for the main section of the tutorial. It's time to integrate some data!

Connect an External Data Source

As an exercise, let's connect an existing OData 2.0 data source that's publicly accessible.

1. From Setup, enter `External Data Sources` in the Quick Find box, then select **External Data Sources**.
2. Click **New External Data Source**.
3. Enter `OrderDB` as the label. As you click or tab away from the label field, the name field defaults to OrderDB.
4. Select **Salesforce Connect: OData 2.0** as the type.
5. Enter `https://orderdb.herokuapp.com/orders.svc/` as the URL.
6. Leave the remaining settings with their default values and click **Save**.

Note

Because this is a sample, read-only database, no authentication is required. A real external system would likely require some credentials. You can configure Salesforce Connect to use the same set of credentials for all access to the data source or separate credentials for each user.

Now that you've configured an external data source, you can select the tables you wish to integrate into your Salesforce org.

Create External Objects

You can create or modify an external object.

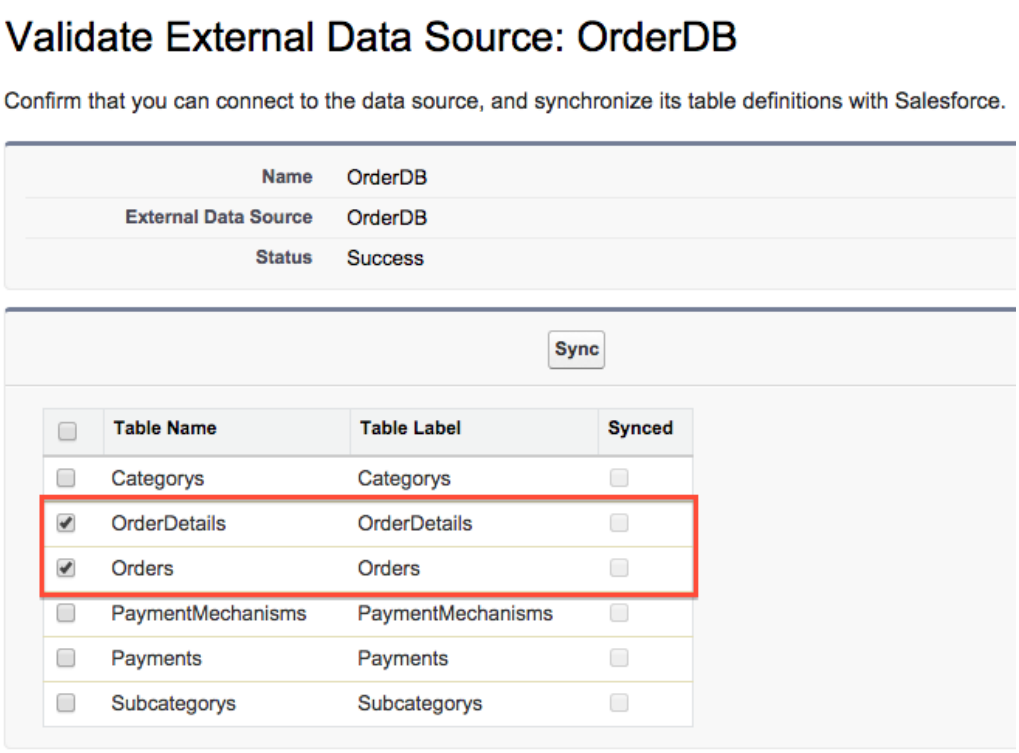
1. From Setup, enter `External Data Sources` in the Quick Find box, then select **External Data Sources** and then click the OrderDB external data source.
2. Click **Validate and Sync**.

Note

Salesforce Connect retrieves OData 2.0 metadata from the sample database and lists the available tables. Click here for a look at the metadata

XML.

3. Select both Orders and OrderDetails.



4. Click **Sync**.

Syncing creates the external objects corresponding to the tables that you selected. Syncing does not store any data in Salesforce. Syncing only defines mappings to external tables or repositories that contain the data. These mappings enable Salesforce to access and search the external data.

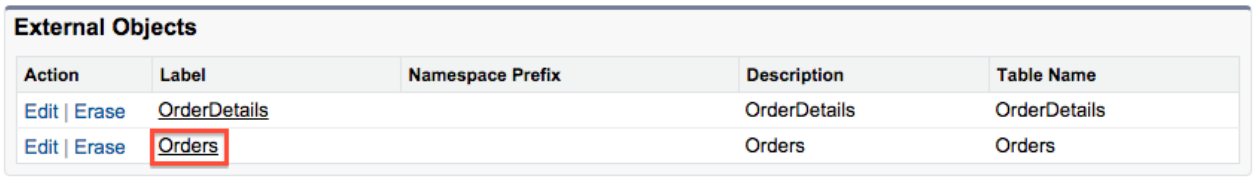
Note

You can choose to manually create the external objects. Doing so enables you to customize the external object names, decide which table columns to create custom fields for, and customize the custom field names. From Setup, enter `External Objects` in the Quick Find box, then select **External Objects**.

View External Data

After you've connected the external data source and defined the external objects, you can view the external data directly in your Salesforce org.

1. From Setup, enter `External Data Sources` in the Quick Find box, then select **External Data Sources**.
2. Click the OrderDB external data source.
3. Scroll down to External Objects and click **Orders**.



Salesforce Connect's sync process created this external object from the external system's schema. If you're familiar with custom objects, you notice that external objects look similar. The sync process also created a set of custom fields just as you create them for a custom object. The key differences between external object and custom object definitions are:

- External object API names have the suffix __x rather than __c.

- External objects have a reference to their external data source and a table within that source.
- External objects have different standard fields. **Display URL** is the OData 2.0 URL representing a record in the external database, while **External ID** is the primary key value for each record.

External Object Definition Detail Edit Delete

Singular Label	Orders	Description	Orders
Plural Label	Orders	Name Field	External ID
Object Name	Orders	Deployment Status	In Development
API Name	Orders__x		
External Data Source	OrderDB		
Table Name	Orders		
Display URL Reference Field			
Created By	Pat Patterson, 2/6/2015 5:27 PM	Modified By	Pat Patterson, 2/6/2015 5:27 PM

Standard Fields Standard Fields Help

Action	Field Label	Field Name	Data Type	Controlling Field	Indexed
	Display URL	DisplayUrl	URL(1000)		
	External ID	ExternalId	External Lookup		

Custom Fields & Relationships New Custom Fields & Relationships Help

Action	Field Label	API Name	Data Type	Indexed	Controlling Field	External Alias	Modified By
Edit | Del	customerID	customerID__c	Number(18, 0)			customerID	Pat Patterson, 2/6/2015 5:27 PM
Edit | Del	orderDate	orderDate__c	Date/Time			orderDate	Pat Patterson, 2/6/2015 5:27 PM
Edit | Del	orderID	orderID__c	Number(18, 0)			orderID	Pat Patterson, 2/6/2015 5:27 PM
Edit | Del	shippedDate	shippedDate__c	Date/Time			shippedDate	Pat Patterson, 2/6/2015 5:27 PM

Let's create a custom tab to easily access the order records.

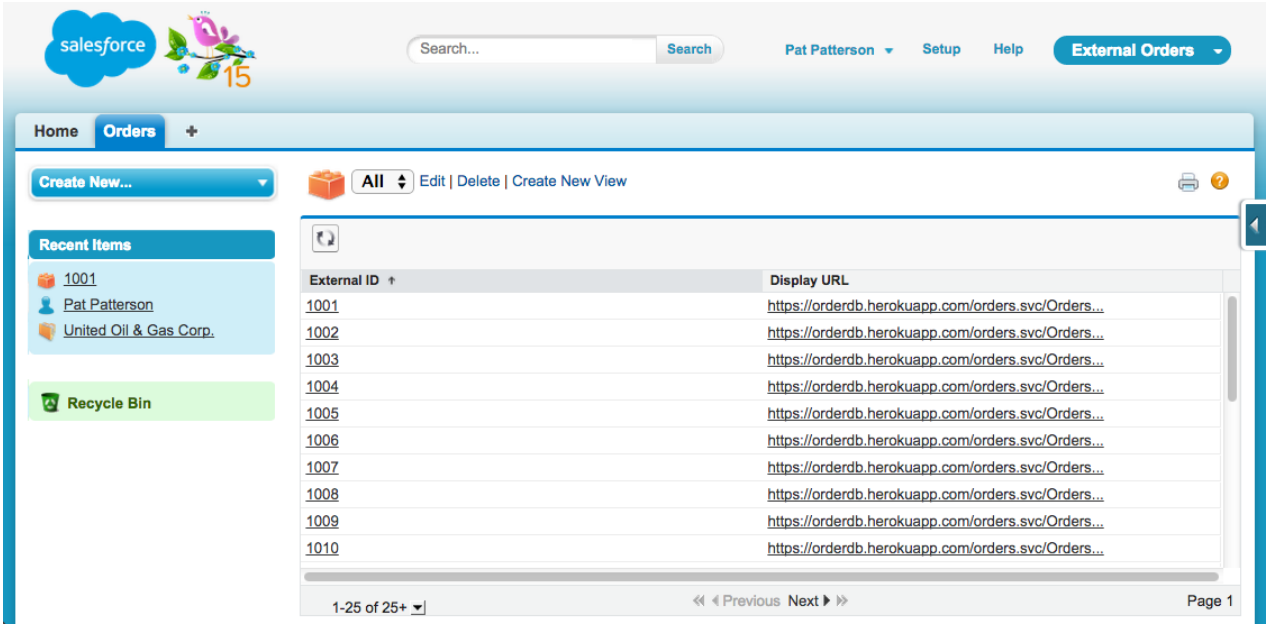
1. From Setup, enter `Tabs` in the Quick Find box, then select **Tabs**.
2. Click the **New** button next to Custom Object Tabs.
3. Select **Orders** as the Object.
4. Click the selector next to Tab Style and choose whichever style you like.
5. Click **Next**.
6. Click **Next** to accept the default tab visibility settings.
7. Click the checkbox next to Include Tab to deselect all the apps.
8. Click the checkbox next to External Orders to select it.
9. Click **Save**.

Note

Because there's also a standard object named Order, you now have two tabs with the label of Orders. You can change the tab name for your external object by changing the label in the object definition.

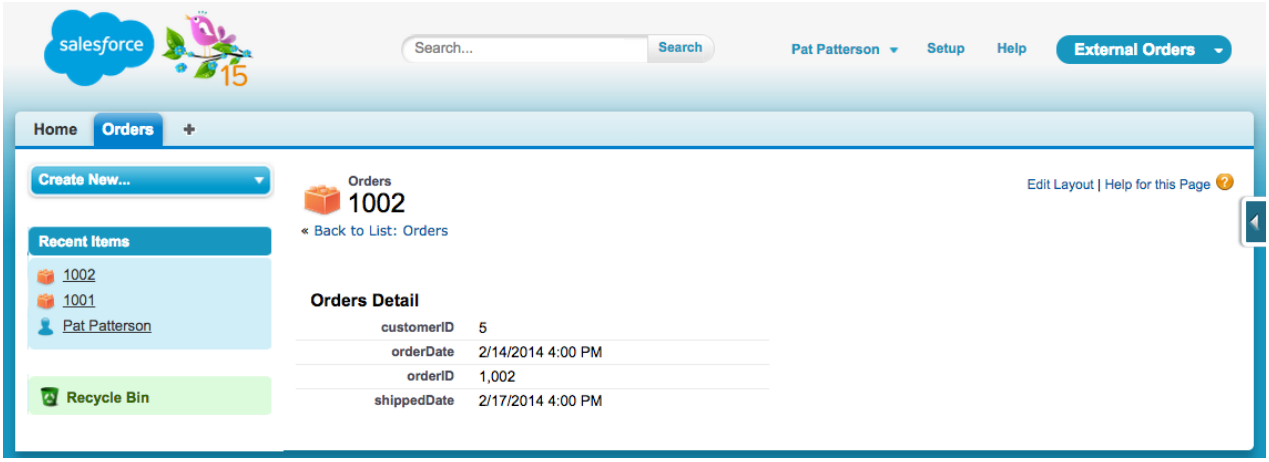
You can now view the external order data as if it was stored in custom objects in your Salesforce org.

1. If the app menu (top right) is not already showing External Orders, click the app menu and select it.
2. Click the **Orders** tab.
3. Click the **Go!** button next to View: All.



Salesforce Connect retrieved order IDs for the first 25 order records from the sample order database. You can configure the fields you want displayed, as in any other list view.

4. Click one of the order External ID values.



Salesforce Connect retrieved all the field values for the order you selected.

It's important to remember that external data is never duplicated in your Salesforce org. Salesforce Connect always fetches current data from the external system in real time.

Now that you can see external data in your org, you can also link it to existing data by creating lookup relationships. You'll see how to do that in a later unit.

Set Up User Authentication

The sample database used in this unit did not require authentication. However, a real external system is likely to require login credentials. You have two options for setting up user authentication for an external data source.

- **Named Principal**—Your entire Salesforce org shares one login account on the external system.
- **Per User**—Your org uses multiple login accounts on the external system. You or your users can set up their personal authentication settings for the external system.

For more information, see Identity Type for External Data Sources in the Salesforce Help.

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Integrate External Data

Learning Objectives

After completing this unit, you'll be able to:

- Configure an external lookup relationship.
- Configure an indirect lookup relationship.
- Customize the display of external data.
- Enable record feeds for an external object.
- Show external data in the Salesforce1 app.

Define Relationships for the External Objects

After you've configured an external data source and defined external objects, you can integrate the external data into Salesforce using relationship fields. You can define three types of relationships for external objects.

- **Lookup relationship**—Links a child standard, custom, or external object to a parent standard or custom object. You can only use this type of relationship if the external data includes a column that identifies related Salesforce records by their 18-character IDs. If that's not the case, use one of the following two types of relationships, which are unique to external objects.
- **External lookup relationship**—Links a child standard, custom, or external object to a parent external object. The values of the standard External ID field on the parent external object are matched against the values of the external lookup relationship field. For a child external object, the values of the external lookup relationship field come from the specified External Column Name.
- **Indirect lookup relationship**—Links a child external object to a parent standard or custom object. You select a custom unique, external ID field on the parent object to match against the child's indirect lookup relationship field, whose values are determined by the specified External Column Name.

This table summarizes the different types of external object relationships.

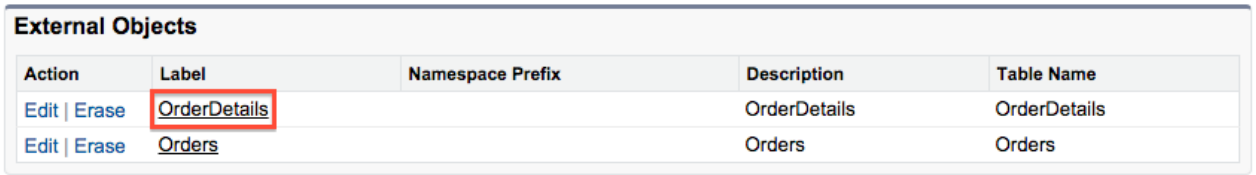
Type of Relationship	Child Object	Parent Object	Must External Data Contain Salesforce IDs?
Lookup	Standard, Custom, or External	Standard or Custom	Yes
External Lookup	Standard, Custom, or External	External	No
Indirect Lookup	External	Standard or Custom	No

As an example, let's see how you can configure lookup relationships to link orders to their line items and to accounts in your Salesforce org.

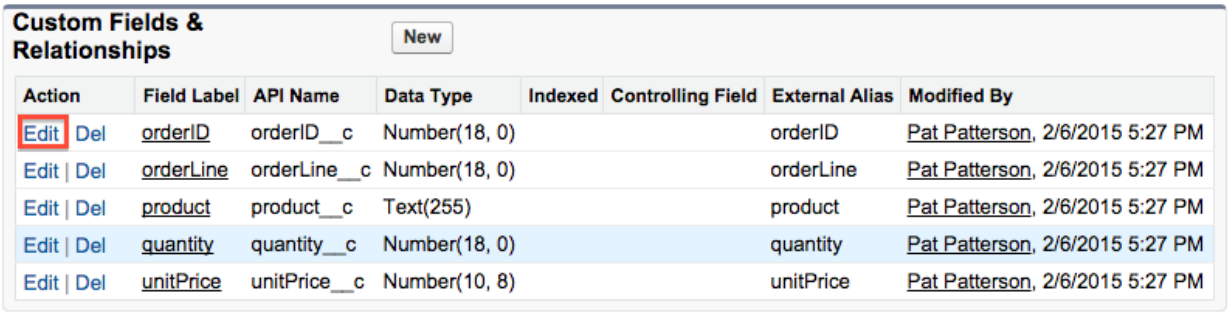
Configure an External Lookup Relationship

In the previous unit, you were able to view external order data in your Salesforce org. Recall that when you selected the Orders table for syncing, you also selected the OrderDetails table, which contains line items for each order. By creating an external lookup relationship from OrderDetails to Orders, you can see the line items on an order's page in your org. In effect, you're telling Salesforce that a field on one object (OrderId on OrderDetails) corresponds to the external ID field on an external object (Orders).

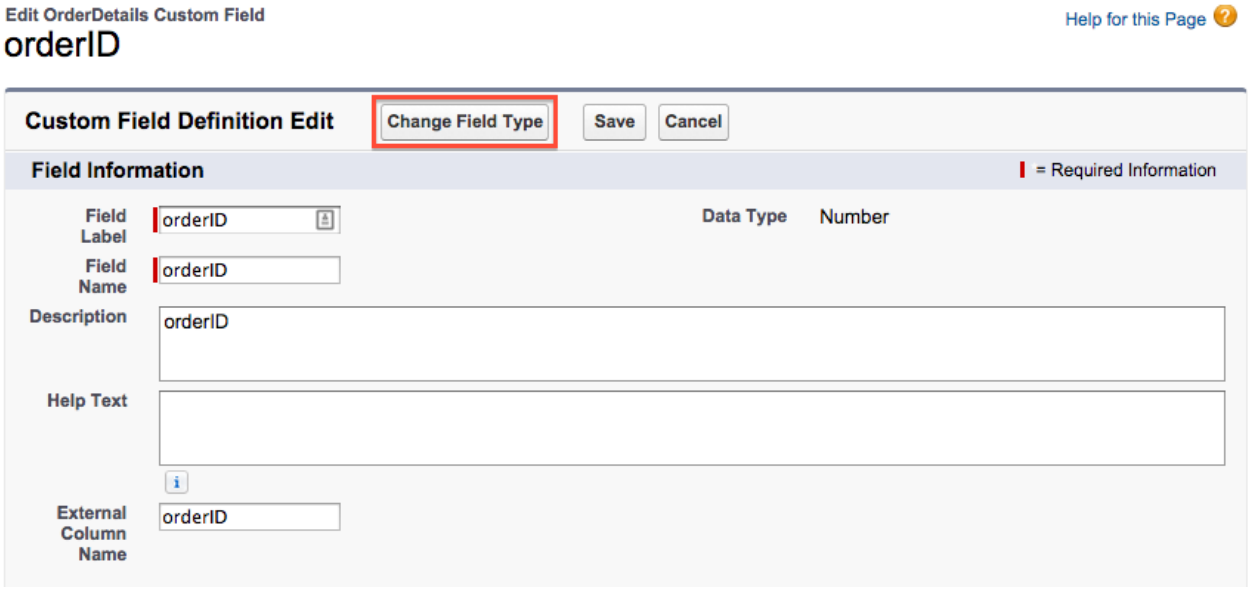
1. Log in to your Salesforce Developer Edition.
2. From Setup, enter `External Objects` in the Quick Find box, then select **External Objects**.
3. Click the **OrderDetails** external object.



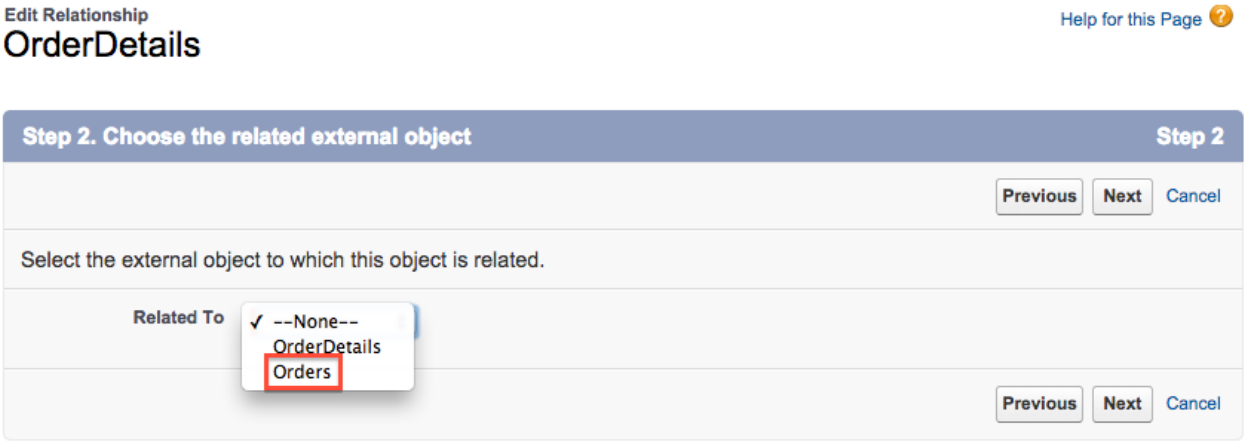
4. Click the **Edit** link next to Order ID.



5. Click the **Change Field Type** button.



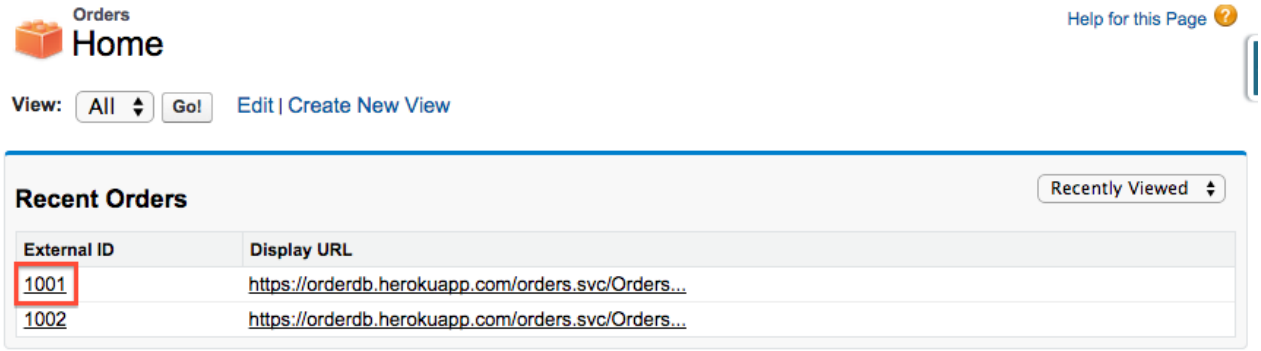
6. Select **External Lookup Relationship** and click **Next**. An external lookup relationship can link any object to an external object.
7. Select **Orders** as the related object and click **Next**.



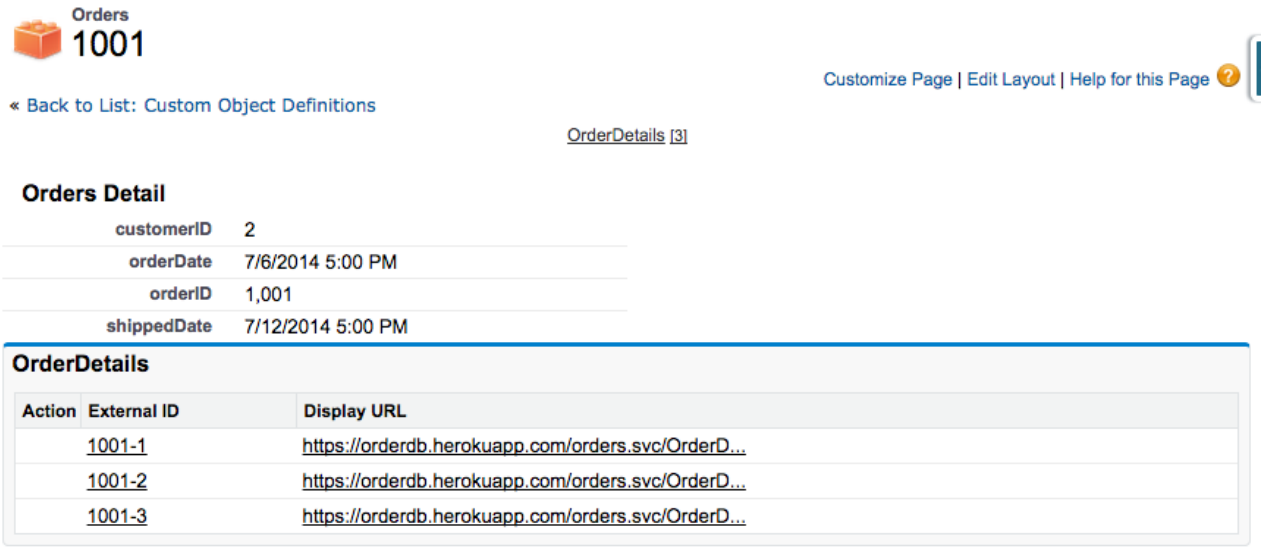
8. Enter `18` as the length and click **Next**.
9. To make the relationship visible to all profiles, select the **Visible** option and click **Next**.

In a real production deployment, you would carefully analyze which profiles should have access to order line items.

10. Click **Save** to accept the defaults—you definitely want an OrderDetails related list on the Orders page layout!
11. If the app menu (top right) is not already showing External Orders, click the app menu and select it.
12. Click the **Orders** tab.
13. Click the External ID of an order in the Recent Orders list.

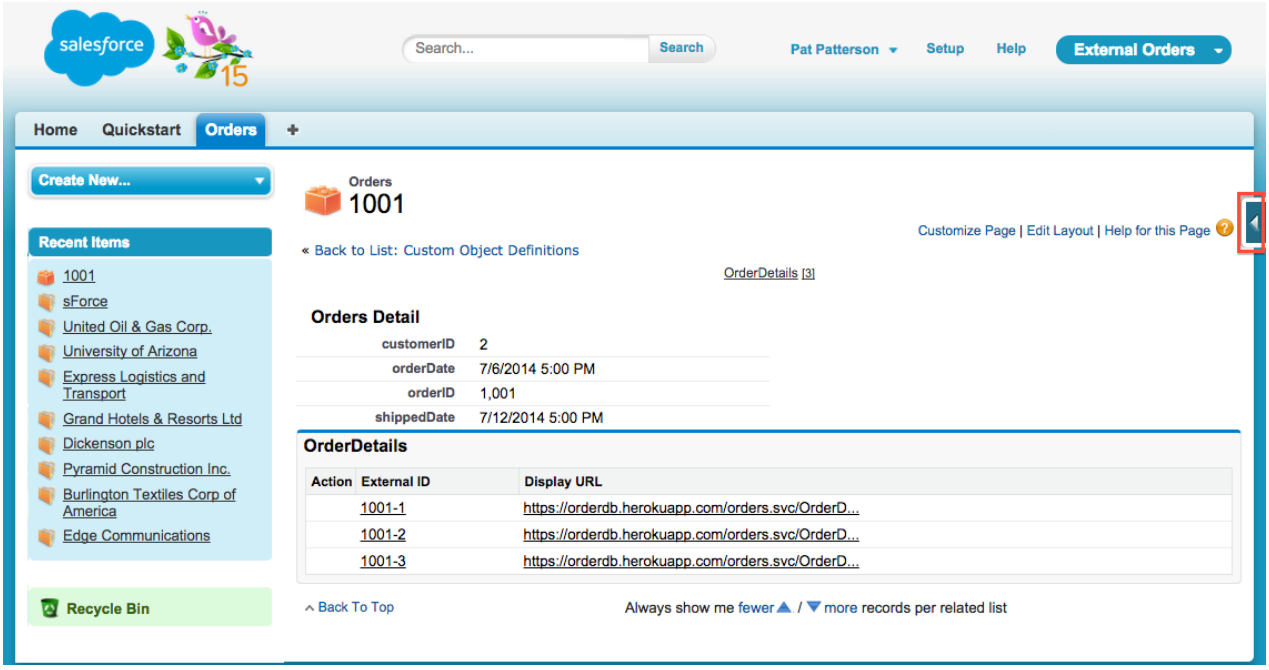


14. Confirm that you see a list of line items for the order.

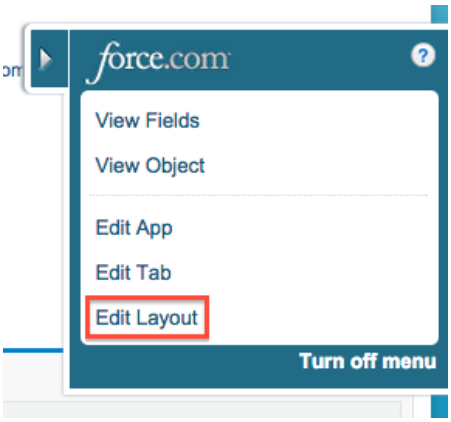


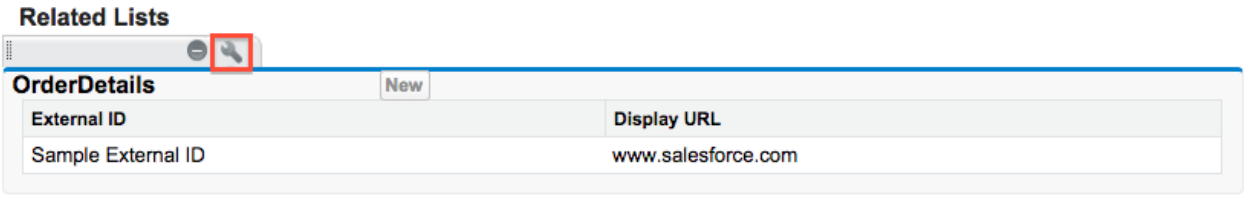
15. You can click a line item's External ID to view its details, but let's show line item details right here on the related list. Select the Force.com Quick

Access Menu by clicking the gray triangle on the right of the page.

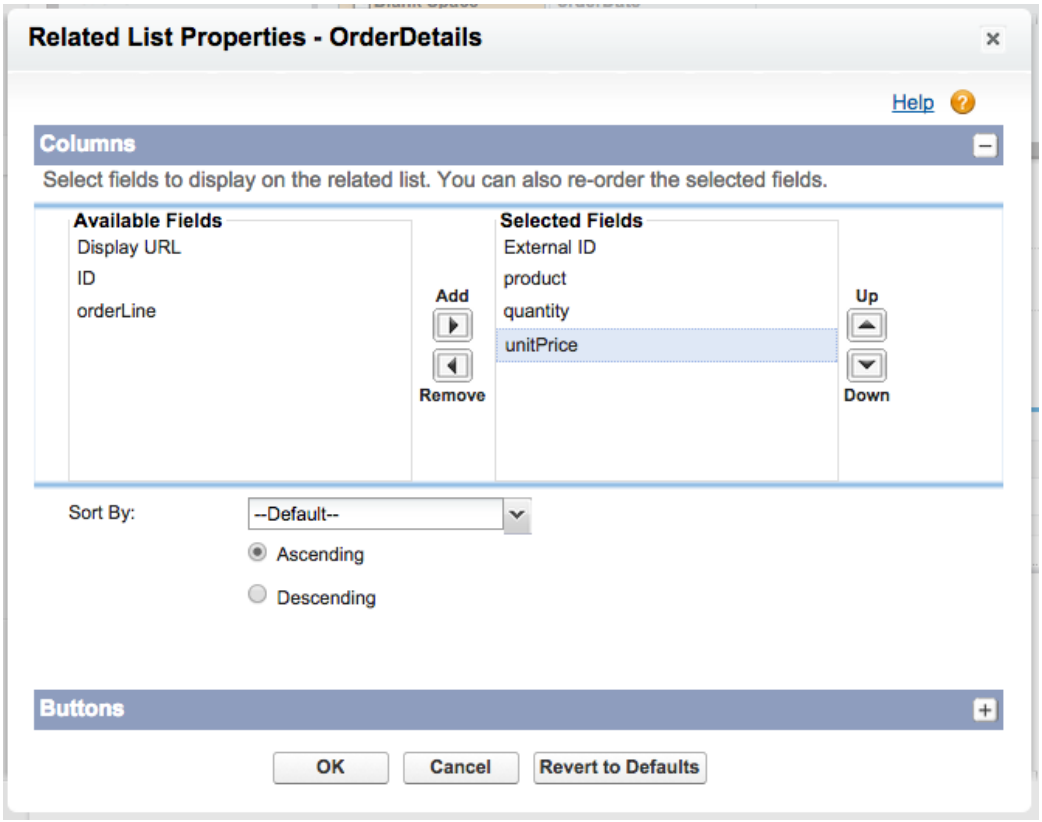


16. Click **Edit Layout**.
17. Scroll down to the OrderDetails related list, and click the wrench icon.





18. Remove Display URL from the Selected Fields, add product, quantity, and unitPrice, and click **OK**.



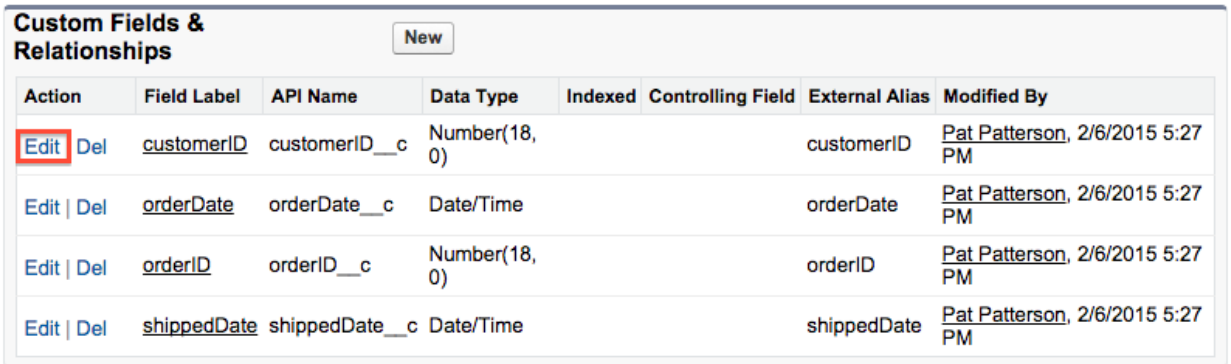
19. Click **Save** at the top of the page. You then see order line item details in the related list.

Configure an Indirect Lookup Relationship

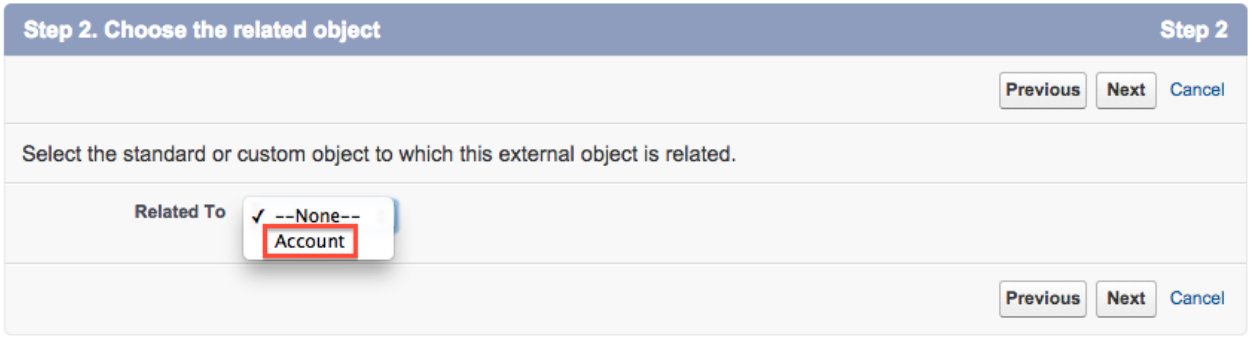
Now that you can see the line items on the order page, the next step is to configure an indirect lookup relationship between orders and accounts. This relationship enables your users to see which account a given order is associated with and all the orders for a given account.

An indirect lookup relationship models a foreign key relationship between an external object and a custom or standard object. This time you're telling Salesforce that a field on an external object (customerId on Orders) corresponds to a unique, external ID field on a custom or standard object (Customer_ID__c on Account). It's an indirect lookup because it references a field other than the standard ID field.

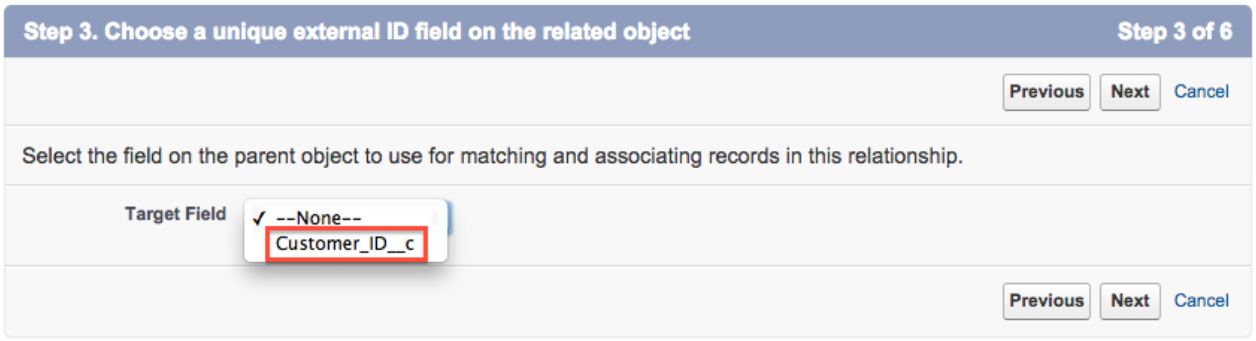
1. If you're not already on the order page, select the **External Orders** app (top right), click the **Orders** tab, and click an order's External ID in the Recent Orders list.
2. Select the Force.com Quick Access Menu by clicking the gray triangle on the right of the page.
3. Select **View Fields**.
4. Click **Edit** next to Customer ID.



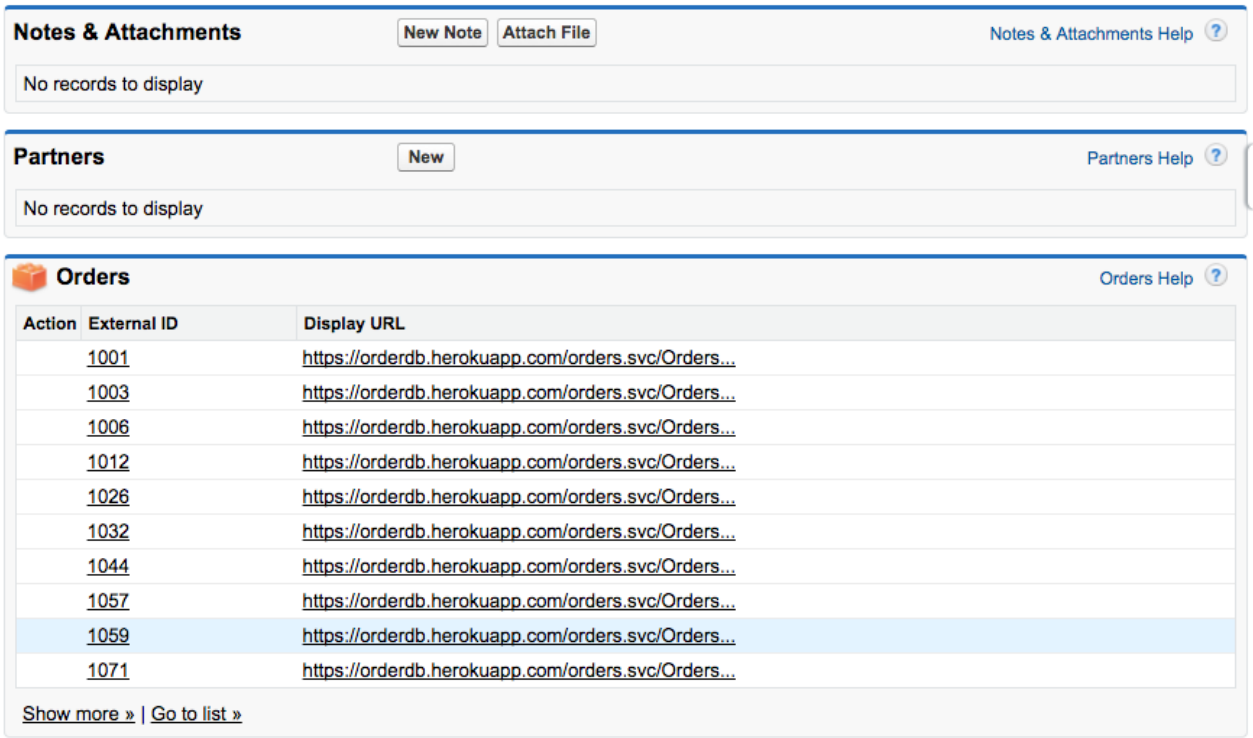
5. Click the **Change Field Type** button.
6. Select **Indirect Lookup Relationship** and click **Next**. An indirect lookup relationship links an external object, such as orders, to a standard object, such as account, or even a custom object.
7. Select **Account** as the related object and click **Next**.



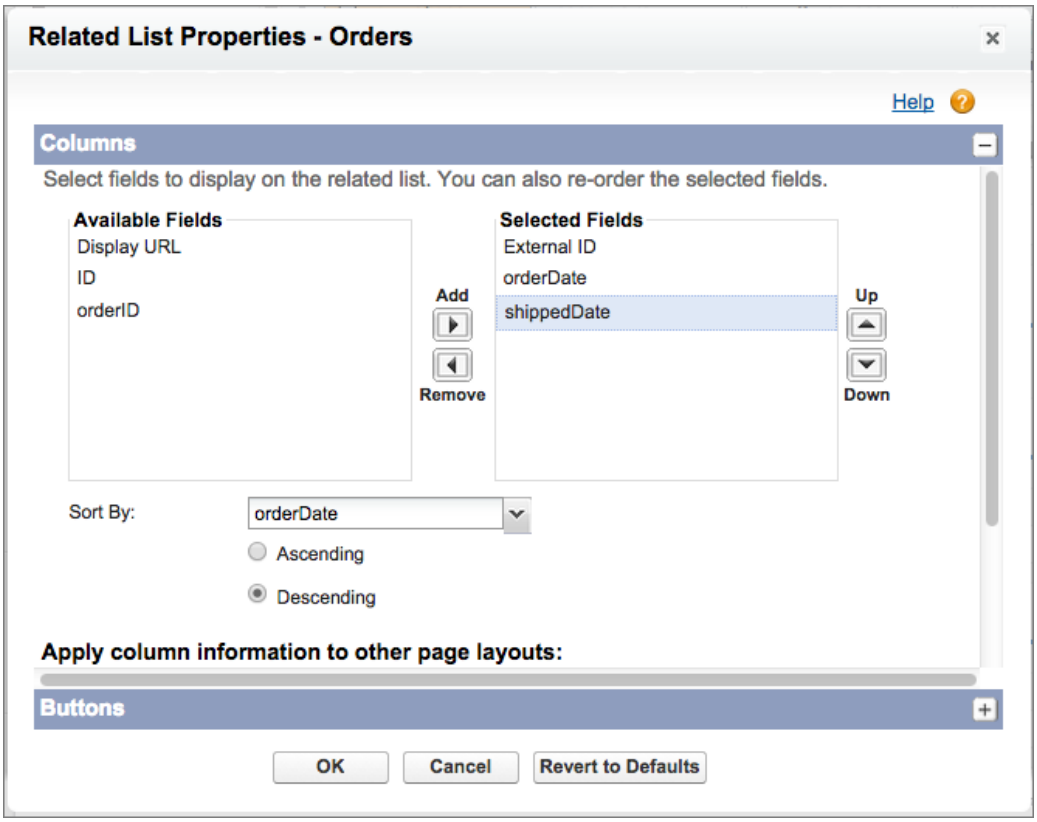
8. Select **Customer_ID__c** as the value of Target Field and click **Next**.



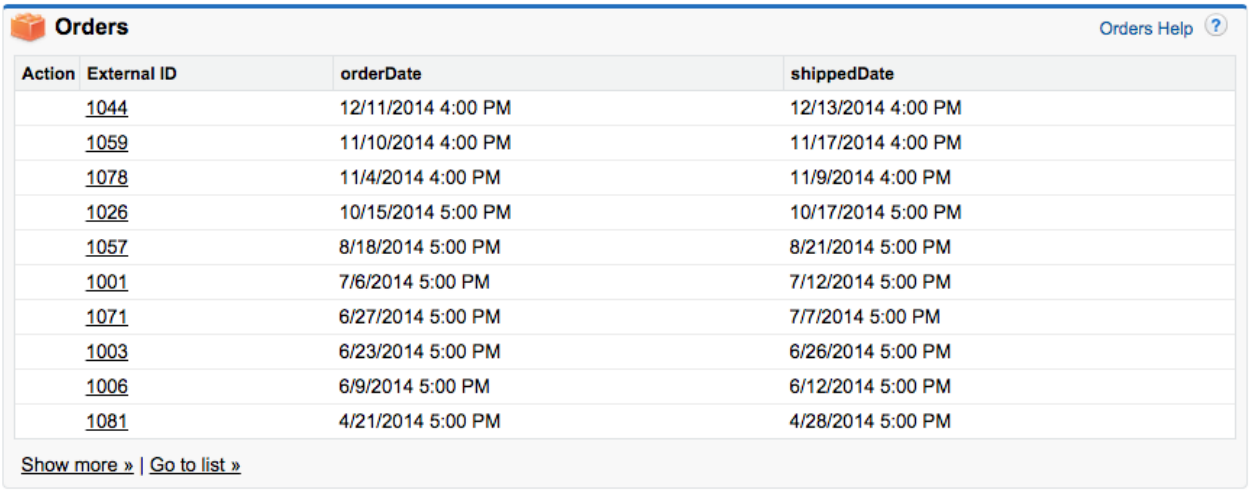
9. Enter `18` as the length and click **Next**.
10. To make the relationship visible to all profiles, select the **Visible** option and click **Next**.
11. Click **Save** to accept the defaults—you want the Orders related list on the Account page layout!
12. If the app menu (top right) is not already showing External Orders, click the app menu and select it.
13. Click the **Orders** tab.
14. Click the External ID of an order in the Recent Orders list.
15. Confirm that the order shows a link in the customerID field.
16. Click the customerID link. You're taken to the corresponding account page. Scroll to the bottom to see a list of orders.



17. Again, you can customize the UI to show more useful information in the related list. Select the Force.com Quick Access Menu by clicking the gray triangle on the right of the page.
18. Select **Edit Layout**.
19. Scroll down to the Orders related list, and click the wrench icon.
20. Remove Display URL from the Selected Fields, add orderDate and shippedDate. For Sort By, click orderDate and select Descending so that you see the most recent orders first. Click **OK**.



21. Click **Save** at the top of the page, scroll down, and see the order dates in the related list.



Action	External ID	orderDate	shippedDate
	1044	12/11/2014 4:00 PM	12/13/2014 4:00 PM
	1059	11/10/2014 4:00 PM	11/17/2014 4:00 PM
	1078	11/4/2014 4:00 PM	11/9/2014 4:00 PM
	1026	10/15/2014 5:00 PM	10/17/2014 5:00 PM
	1057	8/18/2014 5:00 PM	8/21/2014 5:00 PM
	1001	7/6/2014 5:00 PM	7/12/2014 5:00 PM
	1071	6/27/2014 5:00 PM	7/7/2014 5:00 PM
	1003	6/23/2014 5:00 PM	6/26/2014 5:00 PM
	1006	6/9/2014 5:00 PM	6/12/2014 5:00 PM
	1081	4/21/2014 5:00 PM	4/28/2014 5:00 PM

Show more » | Go to list »

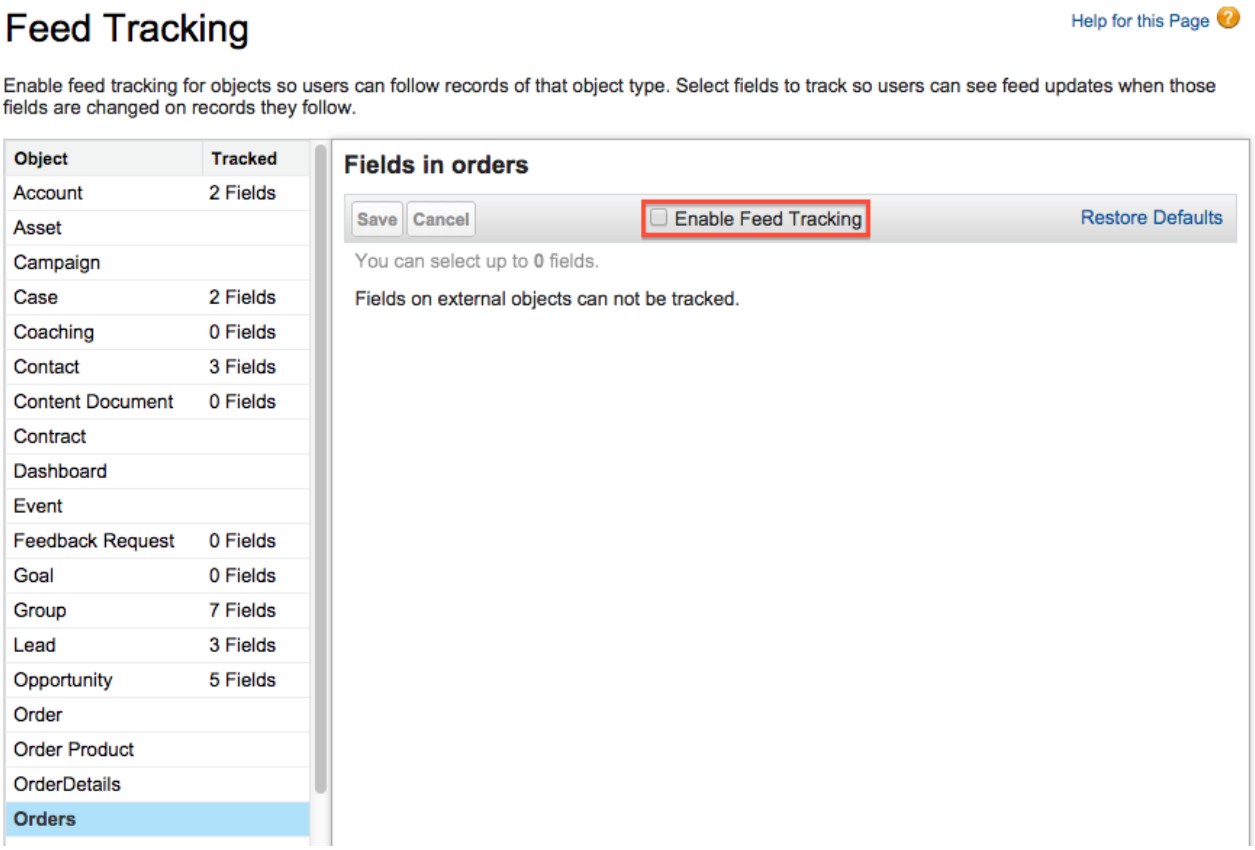
Now your external order data is integrated seamlessly with accounts.

Enable Chatter for External Data

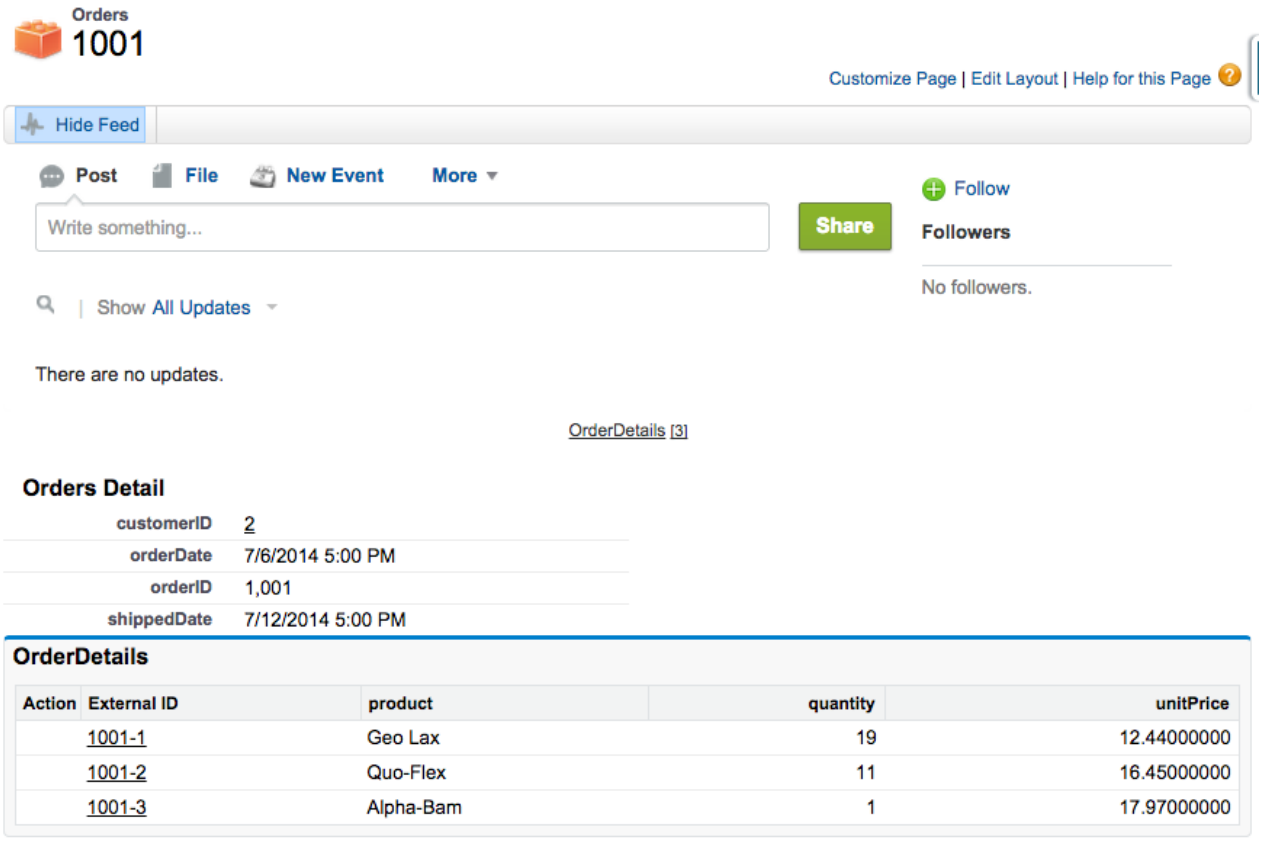
To integrate the external data further into your Salesforce org, let's see how to enable Chatter feeds on order records. In the current release, field tracking is not available for external objects; that is, we cannot configure Salesforce to post to a record's Chatter feed as its field values change, but we can still enable the Chatter feed.

1. Log in to your Salesforce Developer Edition.

2. From Setup, enter `Feed Tracking` in the Quick Find box, then select **Feed Tracking**.
3. Select **Orders** (note the plural; the singular Order is a standard object), click **Enable Feed Tracking** and click **Save**.



4. If the app menu (top right) is not already showing External Orders, click the app menu and select it.
5. Click the **Orders** tab.
6. Click the External ID of an order in the Recent Orders list.
7. Now the order has a Chatter feed. Post some text to the record feed.



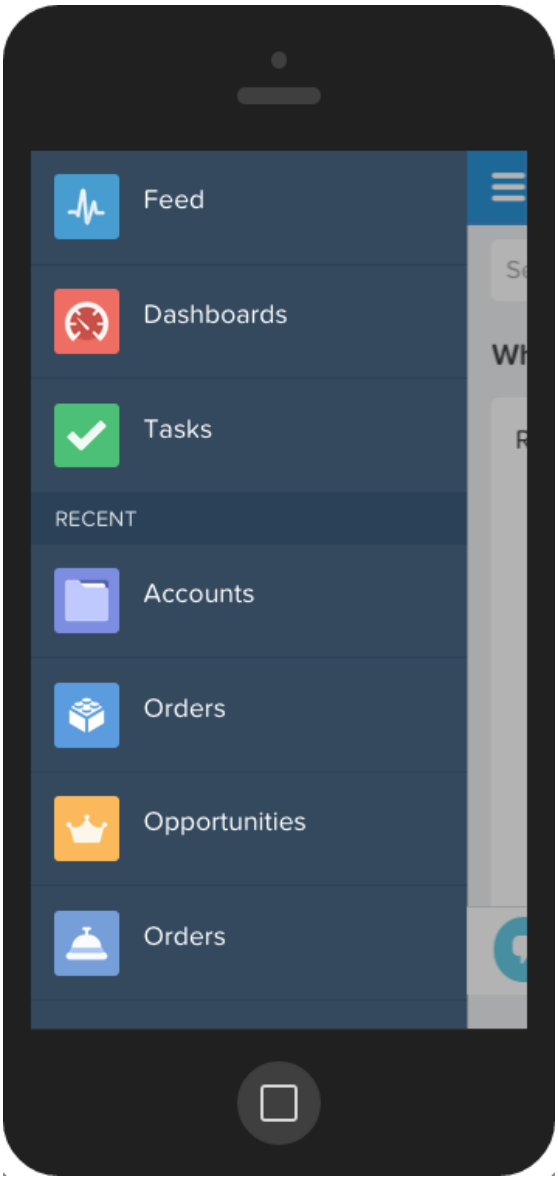
View External Data in Salesforce1

When your external data is available in your Salesforce org, you can view it in the Salesforce1 app.

1. Start the Salesforce1 app. You can run Salesforce1 on your mobile device (get the app) or in the Salesforce1 Simulator Chrome App. Log in with your Developer Edition username and password, if necessary.
2. Tap ☰ to open the navigation menu. Accounts and Orders appear at the top of the Recent list. If they're not at the top, tap **More** to show all the

objects in the Recent list.

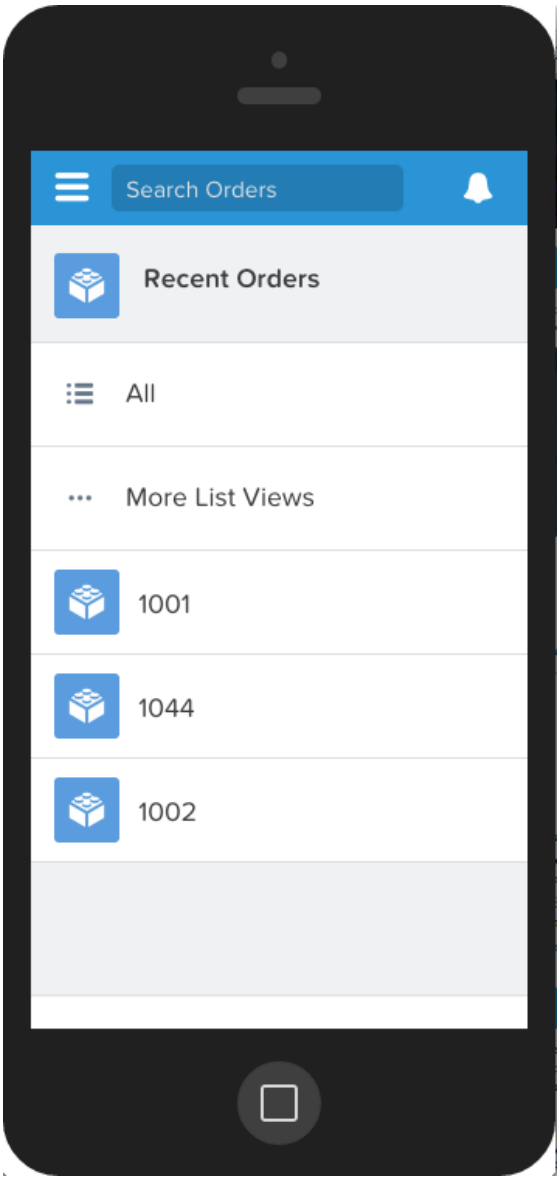
3. Tap **Orders** to see a list of recent orders. In addition to the external object that you have configured, a standard object with the same name, Orders, is listed. If you tap Orders and see no recent records, look for the other Orders object.
4. The order you were viewing in the previous step appears at top of the list. Tap it to view its details.
5. Tap **Feed** to view the order record's feed and your feed post.
6. Tap **Related**, then tap **OrderDetails** to see the order's line items.
7. Tap twice, then tap . Tap **Accounts** to see a list of recent accounts.
8. Tap the top account in the list, then tap **Related**. Scroll down the related lists, then tap **Orders** to see the orders for that account.

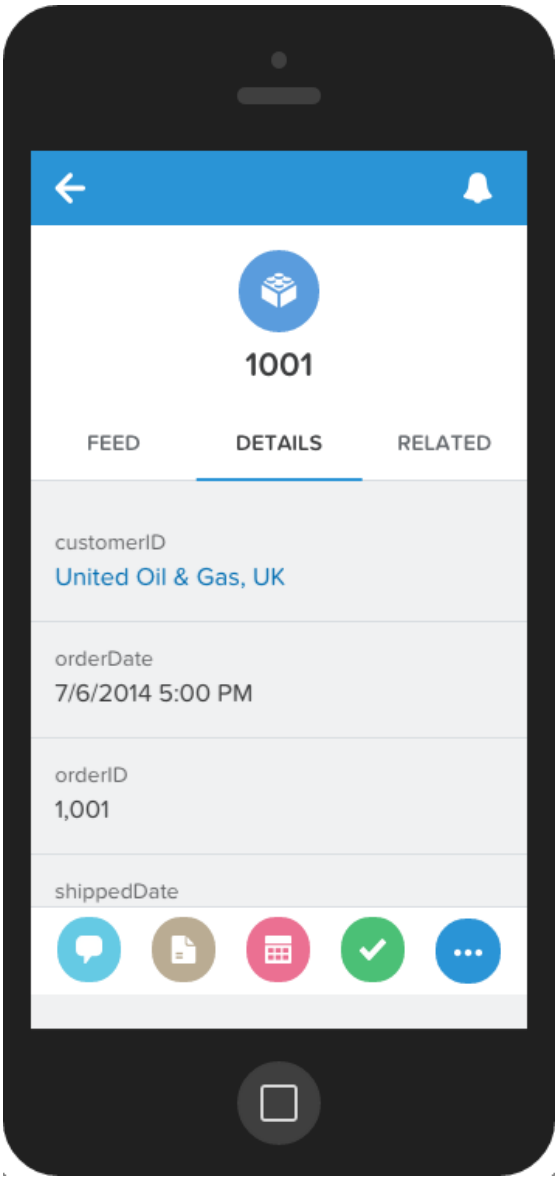


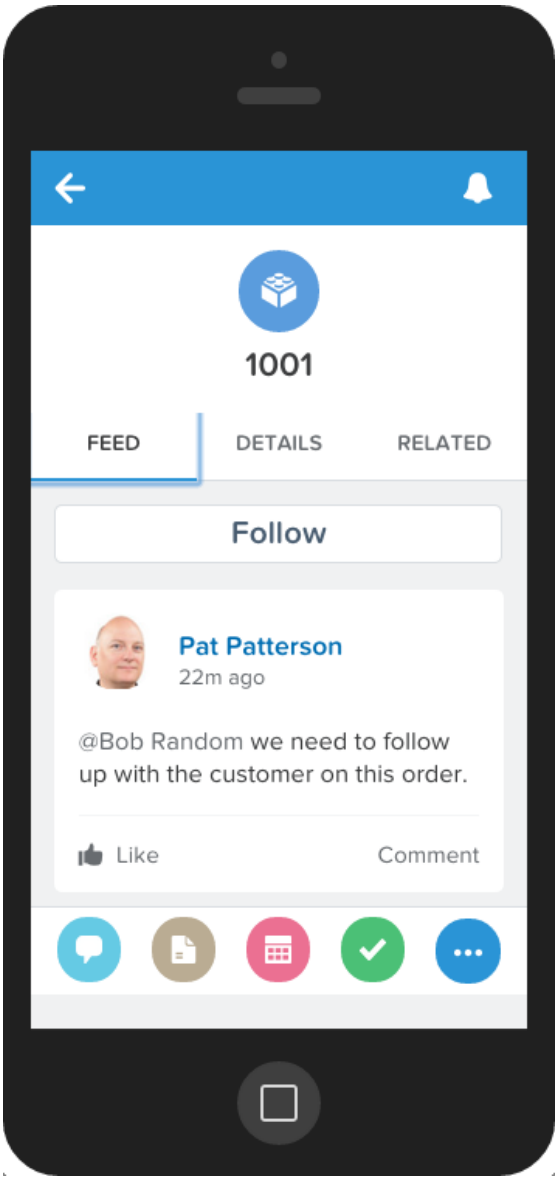
As you can see, your external data is available in Salesforce1 just as it is in the full Salesforce site, with zero extra configuration!

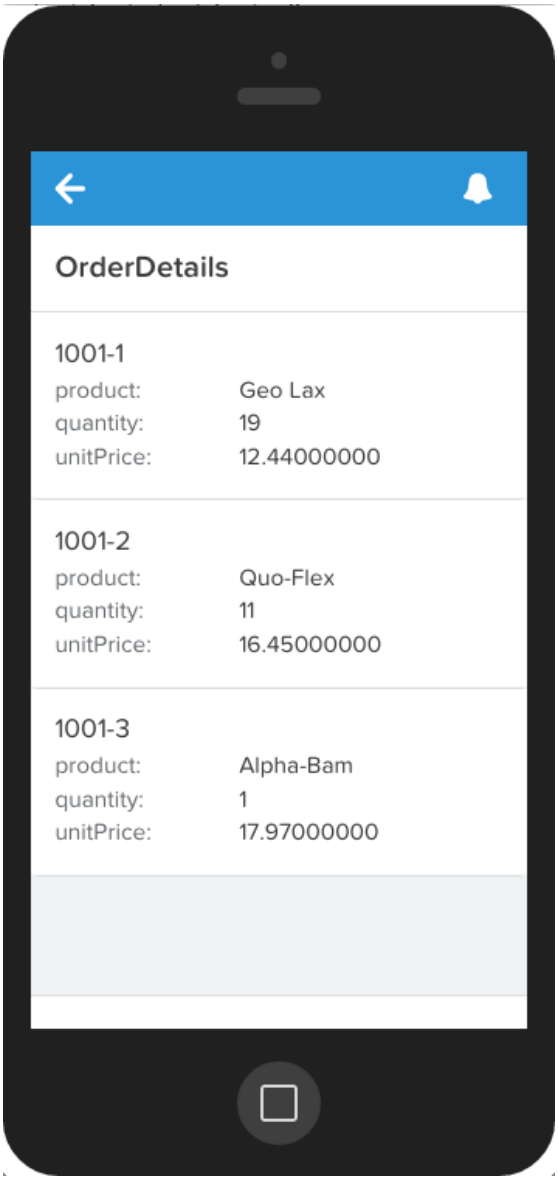
Resources

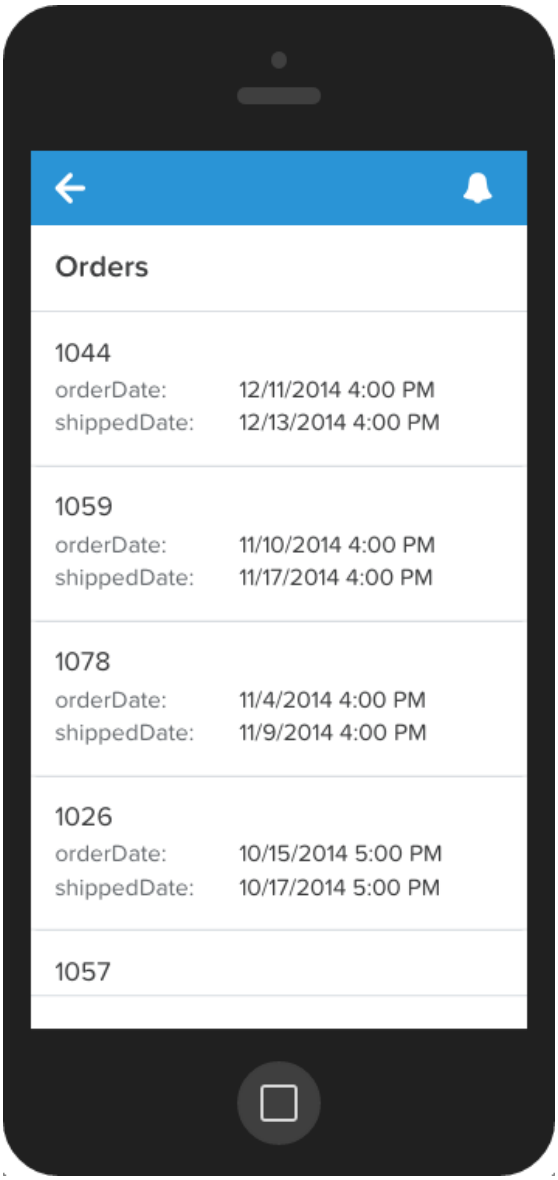
Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.


Search Orders
Recent Orders
All
More List Views
1001
1044
1002


1001
FEED
DETAILS
RELATED
customerID
United Oil & Gas, UK
orderDate
7/6/2014 5:00 PM
orderID
1,001
shippedDate


1001
FEED
DETAILS
RELATED
Follow
Pat Patterson
22m ago
@Bob Random we need to follow up with the customer on this order.
Like
Comment


OrderDetails
1001-1
product: Geo Lax
quantity: 19
unitPrice: 12.44000000
1001-2
product: Quo-Flex
quantity: 11
unitPrice: 16.45000000
1001-3
product: Alpha-Bam
quantity: 1
unitPrice: 17.97000000



Introduction to Application Lifecycle Management

Learning Objectives

After completing this unit, you'll be able to:

- Explain what you can develop in production and when to use a sandbox.
- Explain the key benefits of application lifecycle management.
- Assign roles and responsibilities for the development team.
- Identify the elements of a governance framework.

Application Lifecycle Management

If you've done some of the challenges in Trailhead already, you know it's easy to build and customize apps. For example, the Contact standard object doesn't have a field for Contact Type. You could easily add that custom field in your production org and make the Contact Type field immediately available to all users. But should you?

Asking this question prompts additional questions. For example:

- Will your users know how to use the new field, or will training be needed?

- If the field is required, are there any integrations or import processes which require updating?
- Should the field appear on all page layouts, list views, or in any reports or dashboards?
- Should the field be on the Lead object as well, and if so, does the Lead conversion process require updating?
- Will this field be required for integrations with other systems? This action might require changes to middleware, field mappings, endpoints, and so on.

Non-configuration changes, such as creating views, reports, and dashboards, can be made safely in production. But other changes, such as this hypothetical new Contact Type field, are potential pitfalls of confusion and productivity. Understanding the downstream impact of a change, whether it's preparing for a new user experience or updating an existing integration, are vital steps to minimize disruption and increase feature adoption. But how do you know when to make a change in production versus using sandbox (a separate environment for development and testing)?

The answer depends on your change management strategy. Some companies need the flexibility to make emergency changes in an ad hoc fashion, rather than as part of a scheduled upgrade. Other companies find this disruptive to users, administrators, and developers and opt to roll out their changes with a formalized release process. And some companies do a mixture of both.

In this unit, you learn about the effects of making changes in production, and why at some point you need to separate development from production using a sandbox. You learn what kind of challenges occur in a multi-org environment, and how to create a development strategy to manage those changes.

Note

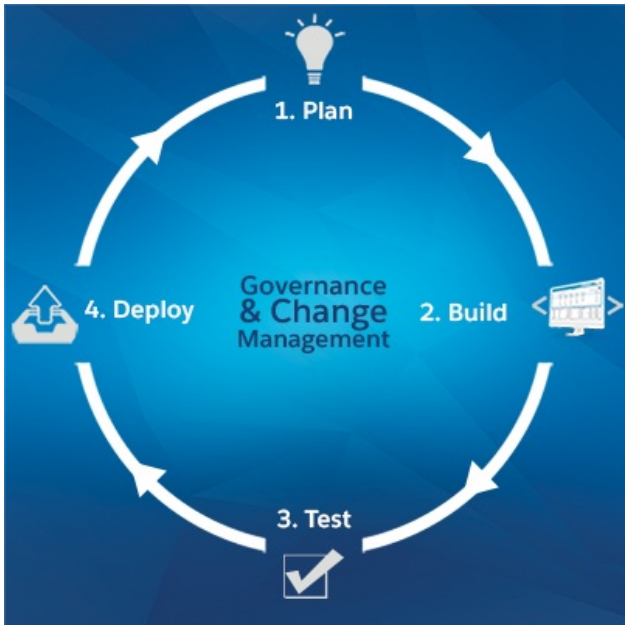
This module does not apply to those creating managed packages for the AppExchange. That development scenario has different best practices.

This Trailhead module is different from the Change Management module. Here's how they're different: If your use cases are simple and you are comfortable with change sets, you might prefer the Change Management module. If you have more complex needs, are an enterprise customer, or need more robust functionality than change sets provide, you might prefer this module. When in doubt, get both badges and decide for yourself!

What Is Application Lifecycle Management?

Application lifecycle management is the process of managing an app's development, from design to final release, and establishing a framework for managing changes. The typical application lifecycle starts with the design of a new app or feature. The app is planned based on requirements analysis and specifications. Next, the app is implemented per the specifications and then tested. The new app is staged for final testing before it gets deployed to production. This cycle repeats for every new app or feature. It's also used for app maintenance, such as when features are enhanced or bugs are fixed. A governance and change management framework directs the development process.

Salesforce Application Lifecycle Management



Step 1: Plan
: An app starts with planning. This step includes requirements gathering and analysis. The product manager creates design specifications and shares them with the development team for implementation.

Step 2: Build
: Administrators and software engineers write the app per the design specifications. The development is done on the Salesforce platform using

declarative tools—the tools available in the user interface—and programmatic tools, such as Apex, Visualforce, and Lightning Components.

Step 3: Test

The app is tested to verify that the changes behave as expected and don't cause undesired side-effects. The quality assurance team first performs unit and functional testing. A small set of experienced people then provide feedback (user acceptance testing). Staging the changes enables final testing before deploying to production.

Note

In Salesforce, development and test environments are represented by sandboxes. A sandbox is essentially a clone of your production org—its features, functionality, user licenses, and setup configuration. Sandboxes are usually used for development, testing, training, and merging changes from other sandboxes.

Step 4: Deploy

When testing is successfully completed and the quality benchmarks have been met, the app can be deployed to production. The release manager manages releases to production. Training employees and partners about the changes is a best practice.

Development Lifecycle Roles

In a smaller company, one person can wear many hats, but in a larger company, specialized roles define what each person is responsible for. Logical roles include the following.

- **Release manager**—Manages the release schedule and coordinates releases with the business. The release manager could be in charge of pulling changes from version control.
- **Product manager**—Provides the business requirements of apps and features, and works with the development team to implement those requirements. The product manager also performs user acceptance testing to ensure that requirements have been implemented.
- **Software developer**—Develops new functionality in sandbox, including both declarative point-and-click development and code.
- **Quality engineer**—Tests new functionality in sandbox.
- **Administrator**—Performs administrative tasks in the production org, and tracks all changes made in production.
- **Trainer**—Conducts training of company employees for new applications and features.

Use Governance to Manage Change

You've learned how to keep your organization lean and clean using the tools that Salesforce provides. However, governance, a method of management, is about more than tools. Governance improves agility by ensuring all members of your team are working together to achieve goals that are aligned with overall business goals.

Three elements of a responsive, adaptable framework for governance are:

Center of Excellence

A few stakeholders from different functional groups work together to ensure that changes support business goals and follow IT best practices and processes.

Release Management

You've already learned how to use tools like change sets and a sandbox to manage changes. If you use a backlog list to manage priorities, you can work on the most important changes first. And if you design and document a complete release management process as you learn more about your organization, everyone who works with Salesforce will be able to know how to do so safely.

Design Standards

Follow key standards for coding, testing, integration, large data volumes, and other areas that affect the services you share with other Salesforce customers.

Create a Release Management Process

By using sandbox and permission sets, you've already created a simple release management process. Add structure by setting up a release schedule and defining criteria for major versus minor releases.

Releases typically fall into one of the following categories:

Daily

Bug fixes and simple changes that do not require formal release management, including reports, dashboards, list views, email templates, and user administration.

Minor

Changes with limited impact, such as a new workflow rule or trigger impacting a single business process. These releases typically require testing, but limited training and change management, and are delivered within a few weeks.

Major

Changes with significant impact, including configuration and code changes with one or more dependencies. Because these releases greatly affect the user experience and data quality, they require thorough testing, training, and careful change management. Major releases typically occur once a quarter (or like Salesforce, three times a year). Releasing on the same day of the week for minor and major releases is a best practice. This allows for company-wide planning and sets expectations with your business users. In addition, don't schedule releases near holidays or other major events.

Your release management strategy evolves over time. Add checks and balances as you discover frequent points of failure or common root causes. The CoE shares these best practices so that everyone who modifies or extends Salesforce knows how to do so safely.

Design Standards

Salesforce is configurable, extendable, and supports integration with other applications and services. By following the same design standards, anyone who modifies your organization can ensure that their changes aren't in conflict with another group's changes. In addition, with Salesforce's multi-tenant architecture, employing design standards helps ensure your changes stay within set governor limits.

Some examples of design standards include:

- Standard naming conventions
- Consistently using the Description field
- Bulkified code
- Standard methods for deprecating classes and fields
- Consistent data architecture across all projects

It's important to have design standards for the following areas:

- Coding
- Testing
- Integration
- Handling large data volumes
- Documentation

The architect or architecture team in your center of excellence defines your company's design standards. Publish your design standards, and communicate them to all teams that work on Salesforce projects, and the rest of IT.

Development Lifecycle Environments

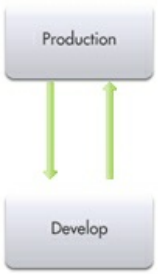
Let's take a look at how to implement a software development cycle in Salesforce, starting with an easy example using a single development and testing environment. Each development and testing environment is represented in Salesforce by a sandbox. You learn more about sandboxes in a later unit.

Single Development and Test Environment

For small or quick projects, it's possible to simplify the development lifecycle by using a single environment for development and testing. If you're just getting started developing in a sandbox, using a single environment is a good way to begin.

This simple development scenario is suitable for small and fast projects, such as:

- New custom objects, tabs, and applications
- Integrations with other systems
- Apps involving Visualforce, workflow, or new validation rules



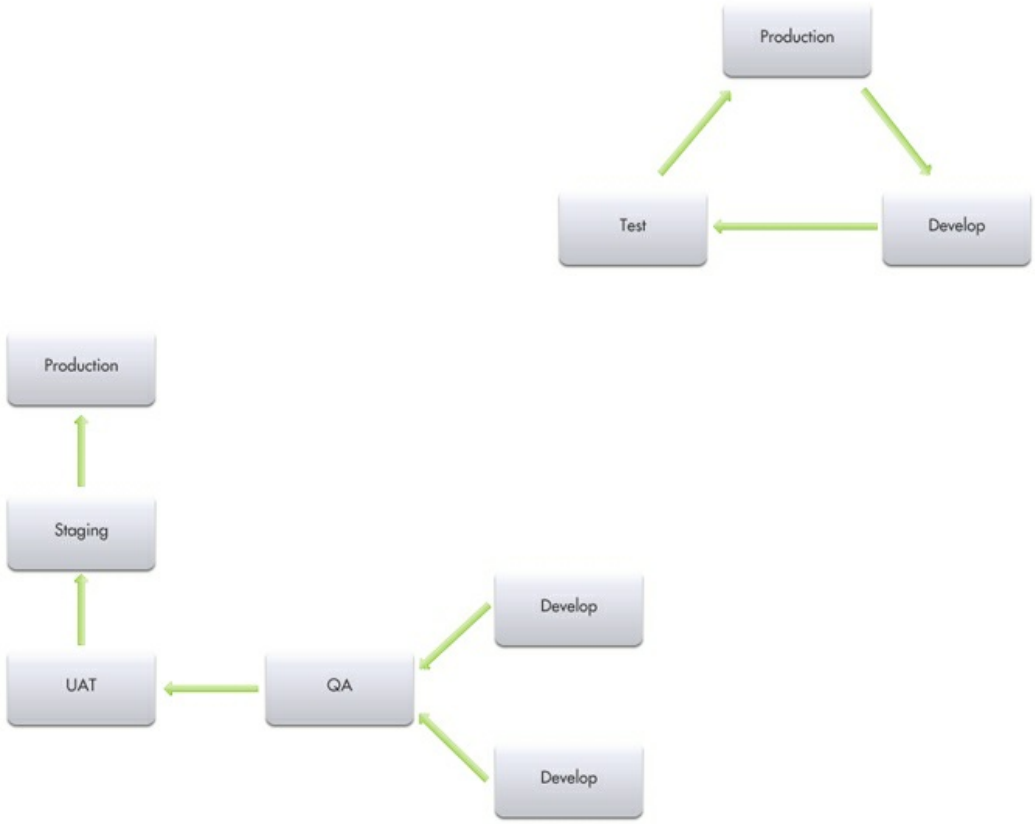
Multiple Development and Test Environments

As your development process matures, you'll find that having multiple sandboxes allows greater flexibility. Even adding a single sandbox for testing frees up the developer sandbox for new features while testing proceeds.

This scenario starts to show its limitations when you have more than one developer working in the sandbox. In short, you can count on conflicts and overwriting changes. In this case, it can be better to have each developer work in their own Developer Edition org and deploy to the sandbox as a means of integration.

When you have multiple developers in multiple sandboxes, you need to merge changes between the sandboxes before you deploy, which requires an integration sandbox. As you add more sandboxes to separate development projects, testing, and deployment, the picture becomes more complete. This next diagram shows the environments used for a development project slightly more complex than the previous one. The team has two developers, each with a sandbox. The changes from the developers are merged into a single testing environment (QA). After testing is completed, the changes are pushed to an environment where staff other than engineers can perform user acceptance testing (UAT). For example, product managers can use the UAT environment to ensure that the features work as expected and demo them. Finally, to ensure a smooth release to production, the changes are first staged

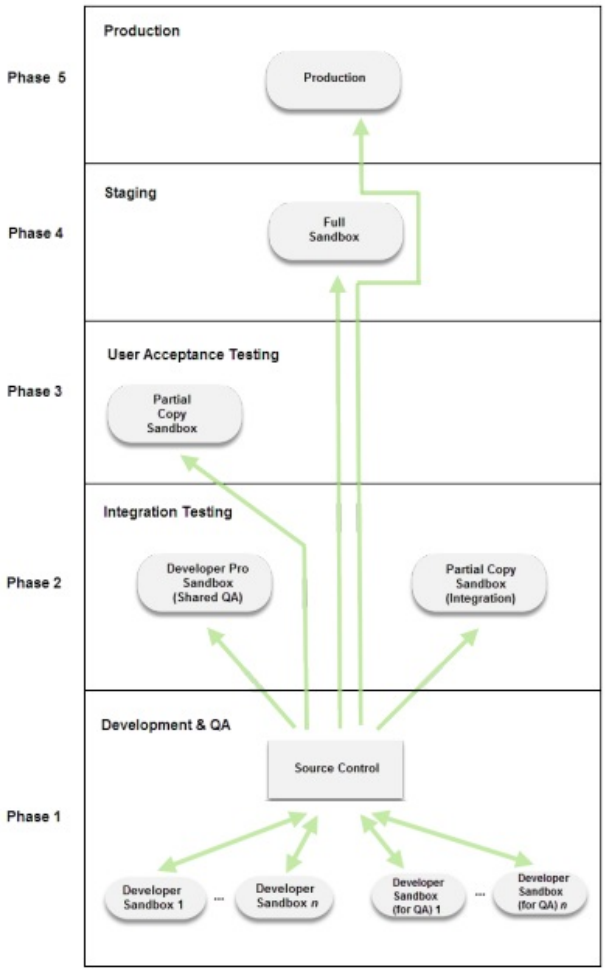
in the staging environment and then released.



Alternatively, changes from multiple developers can be merged in an external source control system. The source control system hosts the metadata (which includes source code) of projects in development in separate branches. When the app is deployed to production, it's deployed from the source control system. The following diagram shows a comprehensive release cycle with all the intermediate development, testing, and staging environments. This release cycle uses a source control system.

Use a Central Source Control Repository with Multiple Environments

Use a source control repository, such as git, for team development. Using an external source control repository enables merging changes from multiple developers. Also, the repository allows isolation of projects that are in development. The source control repository is the source for all customizations deployed to all other environments. After the changes have been tested and validated in the various environments, a test deployment is made to the staging environment. If the deployment succeeds, the changes are migrated to production. This next diagram shows the various environments used in a team development project. The diagram is broken up in the different phases of the development lifecycle for releasing an app.



Resources

Learn More About Sandboxes

Learning Objectives

After completing this unit, you'll be able to:

- Choose the appropriate sandbox for different scenarios.
- Know when to use sandbox templates and when to import data.
- Manage users and licenses in sandbox.

Sandboxes

Sandboxes create copies of your Salesforce org in separate environments. Use them for development, testing, and training, without compromising the data and applications in your production org. Sandboxes are isolated from your production org, so operations that you perform in your sandboxes don't affect your production org.

Note

Where's my sandbox? Your Trailhead Developer Edition org doesn't contain a sandbox because sandboxes aren't available in Developer Edition orgs. Sandboxes are available in some editions, including Professional, Enterprise, and Unlimited. The following sections help you understand sandboxes so that you can learn how to use them in a supported edition.

Types of Sandboxes

There are four kinds of sandboxes.

Developer
: Developer sandboxes copy only the org's configuration, no data. You can create or load up to 200 MB of data, which is enough for many development and testing tasks. You can refresh a Developer sandbox once per day.

Developer Pro
: A Developer Pro sandbox can store up to 1 GB of data (about 500,000 records). It's otherwise similar to a Developer sandbox.

Partial Copy
: A Partial Copy is a Developer sandbox, plus a sampling of data that you define in a sandbox template. You have limited control over the data that is copied. You can choose the objects, but not the records to pull. The sandbox can include up to 5 GB of data, which is about 2.5 million records, with a maximum of 10,000 records per object. You can refresh a Partial Copy sandbox every five days.

Full
: A copy of your production organization and all its data. Because the Full sandbox is an exact copy, the amount of data in the sandbox is the same as your production org. You can refresh a Full sandbox every 29 days.

Refresh Sandboxes

Refresh your sandbox periodically to update it with data and metadata components from production. Refreshing sandboxes ensures that your sandbox doesn't miss any changes that were deployed to production by another sandbox or manually introduced in production. When you refresh a sandbox, Salesforce rebuilds your environment and erases its old contents. The content in your new sandbox is replaced with a copy of the production org. Beware that it's possible to overwrite all the development you've done in your sandbox. Therefore, it's a good practice for each developer to have their own sandbox and make their own backups. Establish a process with the rest of the team for refreshing sandboxes.

How often you can refresh your sandbox depends on the sandbox type. The following table lists the refresh cycle per sandbox type, along with some other statistics.

Table 1. Sandbox Comparison

	Developer	Developer Pro	Partial Copy	Full
Refresh Interval	1 day	1 day	5 days	29 days
Copies Data	No	No	Yes	Yes
Size	200 MB	1 GB	5 GB	Same as production
Templates and Sampling	No & No	No & No	Yes & Yes	Yes & No
Bundled Developer Sandboxes	N/A	5	10	15

Sandbox Uses

Full sandboxes are required only for performance and scalability testing, as well as for staging before final deployment. You might also need a Full sandbox to test triggers that make non-selective queries. For all other use cases, a partial copy of data is enough. For testing, it's often better not to copy data at all, but rather load the same set of data every time. Data is time consuming to copy, and a large volume of data can take days to complete.

The following table compares sandboxes and their ideal use.

Table 2. Sandbox Uses

Use Case	Developer	Developer Pro	Partial Copy	Full
Develop	✔	✔	✔	
QA	✔	✔	✔	
Integration Test			✔	✔
Batch Data Test			✔	✔
Training			✔	✔
UAT			✔	✔
Performance and Load Testing				✔
Staging				✔

Note

Creating or refreshing a sandbox doesn't happen via the Metadata API, so the components copied aren't restricted to objects that have XML representations. Therefore, you get a true and complete copy of everything in the production org. After initial creation, you can refresh the sandbox, creating a new sandbox in place of the old one.

Use Data in Sandbox

You can supply data for testing in sandbox in several ways.

- Use sandbox templates to provide real-world data that makes your testing environment more closely resemble your production instance. Sandbox templates are available only in a Partial Copy or Full sandbox.
- Import external data into your sandbox to make repeatable testing easier because the data set is static and can be relied on not to change.

Copy Production Data with Sandbox Templates

If you have a Partial Copy or Full sandbox, you can pick which standard and custom objects to copy over from production by using sandbox templates. Sandbox templates help you limit the size of your sandbox by limiting the data that is copied over. Generally, you need only a representative set of data for testing in sandbox, not all the data from production. Reducing the amount of data that is copied to sandbox can significantly reduce the sandbox copy time.

The sandbox template editor understands the object relationships defined in your org's schema. Some objects are always included because they're required in any org. As you select objects to copy, the editor ensures that the associated required objects are added.

To access sandbox templates:

1. From Setup, enter `Sandboxes` in the Quick Find box, select **Sandboxes**, then click the **Sandbox Templates** tab.
2. Click **New Sandbox Template** or click **Edit** next to an existing template you want to modify.

Import Test Data

You might think it's more convenient to use a Partial Copy or Full sandbox, and simply avoid having to load data, but that's not necessarily better than loading test data. Data on the production org changes all the time, so anything you're repetitively testing can give you unexpected results. Static data provides consistency.

Generally, it's better to load a small, representative set of test data into your Developer and Developer Pro sandboxes. When functionality changes unexpectedly, it's easier to pinpoint where the problem occurred when the data is static. Developer and Developer Pro sandboxes also have a shorter refresh cycle.

Most people load data using CSV files and the data loader. For more information, see Data Loader.

Note

There are other data loaders available on the AppExchange. Some add more functionality and usability over the standard Salesforce Data Loader, and you are encouraged to try them out.

If you're loading large sets of data, you can use the Force.com Bulk API. The Bulk API and supporting web interface allows you to upload and create import jobs, monitor and manage jobs, and receive notifications when the jobs complete. For more information, see Loading Large Data Sets with the Force.com Bulk API.

Manage Sandbox Users and Logins

Anyone with a production login can log in to a sandbox. Although most users in the production org probably aren't aware of the existence of a sandbox, you probably don't want them poking around your development environment either. To manage access, you can deactivate production users in sandbox. That works out well, because the deactivated production users make room for developers, who usually don't have a login to the production system, but need a login to sandbox. This user license trading is necessary, because you have a finite number of licenses.

The easiest way to give developers a login to sandbox is to create them as users in production, but don't activate them until needed. After creating or refreshing a sandbox, deactivate production users who don't need access and then activate the developer users. Just make sure that the user you deactivate in sandbox isn't someone who logs in to that environment.

All licenses are copied from production to sandbox. If a new license is applied in production and you want to apply it to sandbox, you must refresh with the sync tool (see Match Production Licenses to Sandbox without a Refresh). If you don't refresh, you can lose your work.

When you create a sandbox, all email addresses, except the sandbox creator, are modified so that people don't receive emails from sandbox testing.

Note that email delivery defaults to off, so if you need to test automated emails from Apex or workflow rules, you must turn this setting on. Not only are emails not sent, but unit tests fail when `Messaging.sendEmail` throws exceptions.

Resources

Plan Your Production Deployment

Learning Objectives

After completing this unit, you'll be able to:

- Schedule a release to production using a formalized process.
- Use profiles to limit user access during a release window.
- Describe which best practices to follow when changes are made in production instead of sandbox.
- Explain what a metadata type is.
- Describe how to track changes manually when changes are not supported in Metadata API.
- Choose the right tool to move changes when sandbox and production are on different releases.

Plan Your Production Deployment

Any time you deploy changes to a production environment, your users are directly affected. It's a good idea to have guidelines for rolling out new functionality.

1. Don't break anything.
    a. Release your production functionality in a test environment first. If you successfully deploy and test in a full-copy sandbox, you can be fairly confident that your deployment to production will succeed.
    b. Back up all your metadata.
    c. Have a backup plan, just in case. One method is to create another sandbox as a backup. The backup sandbox contains change sets and scripts to revert as many changes as possible, plus the manual steps required.
2. Schedule the release.
    a. Create and announce a maintenance window during which your organization is unavailable to most users. Try to make this time during off hours or in periods of low activity.
    b. Use profiles to control maintenance updates.
3. Inform users of every change.
    a. Create detailed release notes that document the new functionality and behavior changes.
    b. Send an email announcing the main features with a link to the release notes.
    c. Create webinars and training sessions to educate users.

Use Profiles to Limit User Access

During a deployment window, you can use profiles to limit end-user access to the production organization.

1. Alert all active users about the maintenance window using the email wizard. From Setup, enter `Mass Email Users` in the Quick Find box, then select **Mass Email Users**.
2. Create a profile to lock users out during the maintenance window by editing the login hours. Be sure that system administrators or integration users have access if they need it.
3. Roll out objects, tabs, and apps to different user profiles if you want to allow some users access for acceptance testing.

If your organization includes many profiles, use the following strategy for setting up a maintenance window.

1. Create a profile named Maintenance with all login hours locked out.
2. Use the Data Loader to extract and save a mapping of users and their user profiles.
3. At the beginning of the maintenance window, use the Data Loader to change all user profiles, except the administrator's, to the Maintenance profile. It is important to leave login access with the administrator. Otherwise, all users could be locked out of the system indefinitely. If integrations are going to run during the maintenance window, also don't lock out the integration user.
4. At the end of the maintenance window, use the Data Loader to reload the users with their original profiles.

Establish a Process for Changes in Production

Ease of development is one of the many strengths of the Force.com platform. However, when it comes to developing enterprise applications, this ease of development can lead to changes occurring on the production organization while applications are being developed in sandbox.

To guarantee a successful deployment to your production organization, it is necessary to move changes from production to development, so that your development environments have the same changes that occurred on production. This may seem like a backwards thing to do, to move modifications from your production organization to your development environments, but it's necessary because migrating from development to production can overwrite the changes already made on production.

When modifications occur on the production organization, you need to create or refresh a sandbox to get the latest changes. Note that you can't refresh the sandbox used by the development team - the sandbox refresh would wipe out their changes! Therefore, tracking changes between the production and development environments, and then merging those differences from production to development, can be a necessary process.

To make these tasks easier, it's a good idea to establish a change process for your production organization. A change process determines what kinds of modifications can take place on your production organization, when they can occur, and who is responsible for making the changes. The change process you adopt depends on the kinds of modifications you require in your production environment. The following list suggests some options for change processes, some of which can be used in conjunction.

- **Allow no changes on production**—This is the simplest and most draconian measure you can enforce. In effect, you are sacrificing immediate setup changes for easier deployment. If you choose this process, all development happens in sandbox. All enhancements would go through a formal request process and would be rolled out simultaneously. You can still carry out ad hoc changes with this policy, but they have to follow the same process as larger scheduled changes.
- **Modify only components that have XML representations**—Manually tracking and deploying changes is far more complicated than using diff/merge tools. If you can limit production changes to components that are accessible through the Metadata API, you can simplify change tracking, merging, and deployment.
- **Allow only one administrator to make setup changes**—Some organizations find it useful to assign a single administrator who is responsible for all setup changes on production. This process makes it easier to track changes on the production organization and replicate those changes back into development environments between sandbox refreshes. This approach is flexible because it allows changes in production and project-based development at the same time. However, it also adds a single point of failure, and that administrator needs to understand the impact of changes to any aspect of the system. The single-administrator approach might be difficult to scale in an enterprise environment.
- **Schedule production changes**—If your organization requires frequent changes to the production environment, you can schedule a time to migrate those changes to your development environments. Depending on the number of organizations you have, this could be an easy migration done frequently (weekly, perhaps), or a more elaborate process performed less often (once per quarter). A regular deployment schedule also helps users' expectations for new functionality and training.

Metadata Types

Customizations and feature settings, such as custom objects or account settings, Apex code, and Visualforce pages, are represented as metadata components. You can retrieve or deploy each metadata component as an XML file with the Metadata API or by using deployment tools, such as the Force.com Migration Tool. The category of a metadata component is a metadata type. For example, three custom object components are represented each as an XML file. The type of each component corresponds to the CustomObject metadata type.

The following XML is a basic representation of a custom object named `Book`. `Book` is the name of the metadata component and its corresponding metadata type is `CustomObject`.

```
<?xml version="1.0" encoding="UTF-8"?>
<CustomObject xmlns="http://soap.sforce.com/2006/04/metadata">
    <deploymentStatus>Deployed</deploymentStatus>
    <label>Book</label>
    <nameField>
        <label>Book Name</label>
        <type>Text</type>
    </nameField>
    <pluralLabel>Books</pluralLabel>
    <sharingModel>ReadWrite</sharingModel>
</CustomObject>
```

You can check in your XML files in a version control system, and you can track changes using the version control system's built-in functionality. Most customizations that you develop in Salesforce, such as custom fields and objects, workflow rules, reports, Visualforce pages, Apex classes, are available through the Metadata API as XML files. You can download the files and work with them in a local file system, which allows you to do many useful things, such as:

- Develop using an integrated development environment, such as the Force.com IDE
- Store and version the metadata files in a source control system
- Copy and paste to create or edit components
- Use tools that implement the Metadata API, such as the Force.com Migration Tool, to send changes from one environment to another

Best Practices for Managing Unsupported Metadata Types

Most customizations and feature settings in Salesforce, such as custom objects or account settings, are available through Metadata API. However, some customizations aren't represented in Metadata API as metadata components. You can't deploy those changes from one environment to another. Instead, you have to manually recreate the changes in each environment.

For example, let's say you are developing an app that uses Opportunity Big Deal Alert, which isn't available as a metadata component. You've developed an Apex class that also depends on this metadata type. Before you can deploy the Apex class, you must create the Opportunity Big Deal Alert functionality in each sandbox and in production. Making manual changes can quickly get complicated with large deployments of many dependent types.

With every release of Salesforce, we support more and more metadata types, and in time, we hope to have everything supported. Although it's not a best practice, the easiest practice is to develop functionality that is represented in the Metadata API whenever possible. As you get more experienced with application lifecycle management, you can introduce more complexity, such as using unsupported types.

Track Changes Manually

You track every change that occurs through the Salesforce user interface. Changes made in the user interface don't appear in your other development environments or version control systems, so a process to refresh these environments is necessary. Concurrent changes between the user interface tools and the Metadata API often create dependencies and are a common source of deployment problems or overwriting changes unintentionally. For this reason, manually track all changes made through the user interface and then have a process for replicating those changes in other environments.

It is important to have a method for tracking all changes, and especially important if those changes require manual migration. A useful tool is a change request form that users and administrators fill out for every enhancement requested or change performed. You could also use a spreadsheet or create a custom application within Salesforce to record change requests and the actual changes made. The tool used is less important than making sure that developers and administrators understand that absolutely every change must be tracked.

Note

The AppExchange includes custom apps that you can install to manage this process, such as the free Change Control app.

Whatever way you use to track changes, list every change that occurred, including:

- Who made the change
- The organization where the change occurred
- Date and time
- Which component was changed
- The deployment order

Use the Setup Audit Trail

Changes made in the Salesforce user interface are logged in the setup audit trail. Using the audit trail in conjunction with your manual change tracking tool is the best way to ensure that you do not miss any changes. How you manage this process is up to you. But it's a good idea to regularly transfer all changes in the audit trail to your change list (once a week, for example) or use the audit trail to cross-check changes on your change list.

Run Tests

Changes made in the user interface can break tests, which then block deployments until those tests are fixed. To avoid delays, run all unit tests frequently in your org.

How Salesforce Releases Affect Your Release Schedule

Salesforce usually upgrades to a new version three times per year. A sandbox preview window occurs right before the upgrade so that you can try out new features before your production organization is upgraded. For a brief time, your sandbox and production organizations could be running different versions of Salesforce.

This preview window can affect your environment in the following ways.

- If your sandbox is on a newer release than production, you can't use change sets to migrate changes. You can use other options to move changes between a preview sandbox and production. For example, you can use the Force.com IDE and Force.com Migration Tool, as long as they are set to an API version that the production org supports.
- If Full or Partial Copy sandboxes have a refresh interval that coincides with a preview window, the sandbox could be in the previous API version. You then need to wait for the refresh interval before you can run the same version as production.

Sandboxes are usually upgraded earlier than the production organization, but it could be later, depending on the instance. The sandbox copy date

determines the schedule. You can view the upcoming maintenance schedule at our trust site by clicking the **View Upcoming Maintenance Schedule** link.

Typically, you create a sandbox to get the latest changes from production. You can deploy and test new features in the sandbox. If everything works as expected, the new features are deployed to production and then to other development environments.

Salesforce upgrades are scheduled during off-peak hours and rarely affect users. However, IT departments often schedule their own batch processes during these same hours, so it's important to avoid conflicts. Things that happen during an upgrade are:

- **New logo**—The Salesforce logo is a quick way to verify which version you're using. Sandboxes can upgrade before or after your production organization. To see if the sandbox has been upgraded, check the logo in the upper left corner of your sandbox home page around the time of a Salesforce release .
- **New features**—Every release contains new features, and consequently, new components available through the Metadata API. Click the What's New link in the Help and Training window to view the release notes.
- **Incremented API version**—The API version increments every release. Access to new features requires connecting to the new version of the API. Having different API versions running in different environments can cause problems if you aren't expecting them.
- **Staggered upgrades**—If your production and sandbox organizations aren't running the same version during the upgrade window, wait to deploy components that are new or have additional metadata. If you migrate changes that have newer components to an organization that hasn't been upgraded to support them, the deployment fails.

Resources

Deployment Tool Options

Learning Objectives

After completing this unit, you'll be able to:

- Choose the appropriate deployment tool depending on your scenario.
- Know when to use managed and unmanaged packages in your organization.

Deployment Tools

You can use various tools to move metadata from one organization to another. It's beyond the scope of this unit to teach how to set up and use each tool. However, at the end of this unit, you'll have an understanding of the available tools, including the recommended tools to use. The Resources section at the end of this unit contains links to videos and documentation that provide more information about setting up and using those tools.

Change Sets and the Force.com Migration Tool are the recommended tools for migration. Change Sets is accessible through the Salesforce user interface and allows migrations between sandbox and production. The Force.com Migration Tool is a command-line tool and migrates data between two environments, including Developer Editions orgs.

The easiest way to move changes between a sandbox and a production environment is with a change set. For small deployments, the UI is easy to use, allowing you to select components and find dependencies. Because everything happens in the cloud, you don't need to bring files to a local file system. You can also reuse a change set. After a change set is locked, you can deploy to all connected environments, secure in the knowledge that nothing can change. You can also clone a change set and make minor changes.

Tip

The Change Management module covers Change Sets.

The Force.com Migration Tool is a Java/Ant-based command-line utility for moving metadata between a local directory and a Salesforce org. Use the Force.com Migration Tool to perform repetitive deployments that can be scripted and that use the same components (the same package.xml file). For example, let's say you make the same deployment to dev, test, and user acceptance testing (UAT) sandboxes and production. You want to be certain that the same components are deployed each time. The reliability of this tool makes it ideal for enterprise customers who are developing complex projects.

The following table compares both tools and provides examples of when it's best to use each tool.

Tool	Point & Click	Scriptable	Best for

Tool	Point & Click	Scriptable	Best for
Change Sets	✔		• Straight sandbox to production migrations • Change management without using a local file system • Auditing previously deployed changes • Enforcing code migration paths • Deploying the same components to multiple orgs
Force.com Migration Tool		✔	• Development projects for which you need to populate a test environment with a lot of setup changes—Making these changes using a web interface can take a long time. • Multistage release processes—A typical development process requires iterative building, testing, and staging before releasing to a production environment. Scripted retrieval and deployment of components can make this process much more efficient. • Repetitive deployment using the same parameters—You can retrieve all the metadata in your organization, make changes, and deploy a subset of components. If you need to repeat this process, it's as simple as calling the same deployment target again. • When migrating from stage to production is done by IT—Anyone that prefers deploying in a scripting environment will find the Force.com Migration Tool a familiar process. • Scheduling batch deployments—You can schedule a deployment for midnight to not disrupt users. Or you can pull down changes to your Developer Edition org every day.

Learn about each tool's considerations to better determine which tool is best suited for your needs.

Change Set Considerations

- You can move metadata only between the production org and its sandboxes. You can't move changes between two production orgs or Developer Editions.
- You can add components with a change set, but you can't delete them. You must use another method to delete components, typically manually.
- Because change sets are cloud-based, they're not ideal when used with a source control system.

Force.com Migration Tool Considerations

- Requires a more developer-oriented skill set, with experience of Ant and scripting tools
- Requires storing the username and password on disk, which some security policies don't permit

Other Deployment Tools and Artifacts

Besides Change Sets and the Force.com Migration Tool, you can use various tools for deployment. The following table compares some of the tools.

Tool	Best For	Limitations
Force.com IDE *Description:* Plug-in to Eclipse used for both development and deployment	• Project-based development • Deployment to any org • Synchronizing changes • Selecting only the components you need	• Some setup required • Not always upgraded at the same time as other Salesforce products • Repeatable deployments require re-selecting components, which can be time consuming and introduce errors
Force.com Workbench *Description:* Lightweight web-based tool that uses your local file system	• Ad hoc queries • Deploy or retrieve components with a package.xml file • Metadata describes • Lightweight data loads	• Not an officially supported product • No project management features

Tool	Best For	Limitations
Force.com CLI *Description:* Command-line interface for Force.com APIs	• Scripted commands and automated tasks • When your security policies dictate that passwords must not be stored on disk; forces interactive login	• Logging in can be difficult behind a firewall

Deployment Artifacts

In addition to deployment tools, packages are another way to move metadata components from one org to another. However, packages are deployment artifacts and not deployment tools. The main use case of packages is for ISV developers to distribute apps to subscribers. But they can be used also for moving changes between Developer Edition orgs.

Artifact	Best For	Limitations
Unmanaged Packages *Description:* A collection of application components that can be distributed and installed in other orgs.	• One-time setup of a development environment • A starting point configuration that can be customized	• You cant't make further changes to packaged components using subsequent packages • Requires a Developer Edition org
Managed Packages *Description:* A collection of application components with a namespace that can be distributed and installed in other orgs. Managed packages can be listed on the AppExchange and are upgradeable.	• Commercial applications • Functionality you want to add in multiple, possibly non-related orgs	• Access to code is limited or hidden • Unique namespace can be bothersome or a blocker • Difficult to modify or delete components • Requires a Developer Edition org

Resources

Move Changes Between Environments with the Force.com Migration Tool

Learning Objectives

After completing this unit, you'll be able to:

- Describe the structure and use of the package.xml file.
- Describe the process of deleting components.
- View the status of a deployment on the Deployment Status page.

Move Changes Between Environments

The process of moving metadata components from sandbox to production is often referred to as deployment. However, the term deployment is also used for a number of similar things, such as:

- Installing an app from the AppExchange and making it available to users
- Changing the operational status of new business logic (such as a workflow rule) from a state of in development to deployed
- Executing the contents of an inbound change set
- Sending metadata from one org to another

For this unit, we're concerned only with the last deployment definition and focus on using the Force.com Migration Tool. With the Force.com Migration Tool, you can move changes between sandboxes, or from sandbox to production.

Tip

You can move metadata components by using other tools, such as Change Sets. Change Sets is covered in Change Management module.

A deployment profile consists of a zip file containing metadata components and a package.xml file. The package.xml file is a manifest that lists everything being sent from the source org to the destination org. There are many metadata types, and you can have many components of a type, so the package.xml file can be long. But the good news is that you rarely have to create the file yourself. Most of the time the package.xml file is created for you. For example, if you're using the Force.com IDE, the package.xml is assembled from the components you select when you create a project.

The exception is if you use the Force.com Migration Tool or Workbench. In this case, you create package.xml files for retrieval (the files you bring to a local file system) and deployment (the files you send back to the server). It's often easier to create a project in the Force.com IDE and then copy package.xml and use the copy for the Force.com Migration Tool.

Note

When using change sets, you don't see the package.xml file. It all happens behind the scenes. When you deploy change sets, the zip file of components and the manifest are handled seamlessly behind the scenes.

Force.com Migration Tool Video

Check out this video to get a quick overview of how to use the Force.com Migration Tool for migrating changes.

Use a Manifest with the Force.com Migration Tool

The Force.com Migration Tool uses a project manifest, a file that's named package.xml. The manifest is an XML file that defines the components you're migrating and specifies the version of the Metadata API to use. Types that are newly supported in the Metadata API can be migrated only in the API version that they were introduced in or a newer version.

Take a look at the following package.xml file. The specifies a metadata type, in this case, `CustomObject`. The is a list of one or more components of that type. In this case, it lists two custom objects, `Invoice__c` and `Line_Item__c`.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package
xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>Invoice__c</members>
        <members>Line_Item__c</members>
        <name>CustomObject</name>
    </types>
    <version>39.0</version>
</Package>
```

To retrieve all members of a type, you use the wildcard (`*`) character instead.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package
xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>*</members>
        <name>CustomObject</name>
    </types>
    <version>39.0</version>
</Package>
```

Note

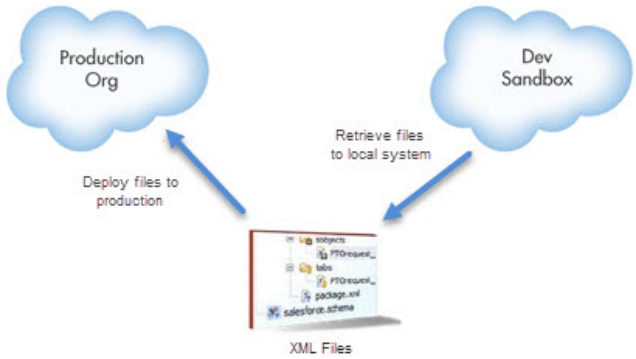
You can't use a wildcard character with some metadata types. For more information, see Metadata Types.

Migrate Metadata Components

Migrating files between environments requires first copying metadata from one org to a local file system. You then transfer the metadata from the local files to another org. Several factors determine how long this operation can take.

How Metadata Files Are Migrated

Migrating changes from one org to another using metadata files requires an intermediate tool that interacts with both environments using the Metadata API. The following figure shows how changes are migrated from sandbox to production.



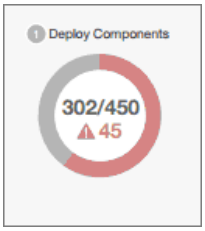
You might be wondering why you need to store files to a local system to migrate files that are stored in the cloud. The Metadata API supports traditional software development tools that operate on source files, such as text editors, diff/merge utilities, and version control systems, all of which require a local file system. After you retrieve the metadata components locally, you can deploy them directly to any other org, including production. You don't need to retrieve the components again for each deployment, unless when you want to get a fresh copy of the metadata.

Monitor the Status of Your Deployments

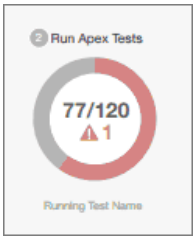
The size and complexity of the metadata components affect the deployment time. To track the status of deployments that are in progress or have completed in the last 30 days, from Setup, enter `Deployment` in the Quick Find box, then select **Deployment Status**. Deployments are listed in different sections depending on their status.

This page lists all deployments—change sets, Metadata API-based deployments, including deployments started from the Force.com IDE and the Force.com Migration Tool, and package installations.

When running a deployment, the Deployment Status page shows you the real-time progress of your current deployment. This page contains charts that provide a visual representation of the overall deployment progress. The first chart shows how many components have already been deployed out of the total and includes the number of components with errors. For example, the following chart indicates that 302 components were processed successfully out of 450 and there were 45 components with errors.



After all components have been deployed without errors, Apex tests start executing, if required or enabled. A second chart shows how many Apex tests have run out of the total number of tests and the number of errors returned. In addition, the chart shows the name of the currently running test. For example, in the following chart, 77 tests have completed execution out of a total of 120, and 1 test failed.



Pending Deployments

You can initiate multiple deployments. Only one deployment can run at a time. The other deployments will remain in the queue waiting to be executed after the current deployment finishes. Queued deployments are listed under Pending Deployments in the order they will be executed.

Deployment Validations

A deployment validation is a deployment that is used only to check the results of deploying components and is rolled back. A validation doesn't save any deployed components or change the Salesforce org in any way. You can determine whether a deployment is a validation only (Validate) or an actual deployment (Deploy) by inspecting the information for pending deployments or the Status column of deployments in the Failed and Succeeded sections.

Delete Components

Administrators can't delete components using change sets. As a result, they might resort to such tactics as deploying empty Apex classes to remove old functionality. This isn't a particularly elegant solution, but it works well enough for admins that don't have other means at their disposal.

To delete components, use the same procedure as for deploying components, but also include a delete manifest file named destructiveChanges.xml listing the components to delete. The file format is the same as package.xml, except that wildcards aren't supported.

You can perform a deployment that only deletes components, or a deployment that deletes and adds components. In API version 33.0 and later, you can specify components to delete before and after other components are added or updated by naming the files destructiveChangesPre.xml and destructiveChangesPost.xml.

The ability to specify when deletions are processed is useful when you're deleting components with dependencies. For example, let's say a custom object is referenced in an Apex class. You can't delete it unless you modify the Apex class first to remove the dependency on the custom object. You can perform a single deployment that updates the Apex class to clear the dependency and then deletes the custom object by using destructiveChangesPost.xml.

Note

In API version 32.0 and earlier, if you specify deletions and additions for the same deployment, the `deploy()` call performs the deletions first.

Resources

Efficient Deployments

Learning Objectives

After completing this unit, you'll be able to:

- Explain the benefits of reducing deployment times.
- Perform a validation and a quick deployment to reduce deployment time to production.
- Describe the benefits of test levels and know which test level to use when deploying components.
- Explain what happens when a Salesforce service update interrupts a deployment.

Efficient Deployments

When you deploy customizations with Apex code to production, local Apex tests are run as part of the deployment. Local Apex tests consist of all tests in the target org that don't originate from managed packages. Testing in deployments helps ensure that the deployed components don't alter or break existing customizations in production. Testing lets you catch and fix issues before the customizations are rolled out to production.

However, if you've been developing many custom apps and your Salesforce org contains thousands of Apex tests, testing can slow down a deployment. Testing constitutes the largest part of total deployment time, and a long deployment slows down your software development lifecycle. When you multiply the deployment time by the number of iterative deployments performed in test environments, the delay becomes even longer. Another disadvantage of long deployments is that they could occasionally be interrupted by Salesforce service updates.

For these reasons, Salesforce provides deployment features that make deployments more efficient. Quick deployments cut down on deployment time by running tests as part of validations. Test levels give you the power to decide how many tests to run in the target org.

Quick Deployments

Roll out your customizations to production faster by running tests as part of validations and skipping tests in your deployments. Because quick deployments don't run tests, they're faster than regular deployments. A quick deployment enables you to schedule deployments to production with more confidence because you know the deployment will succeed and requires a shorter window of time. A Salesforce service update is also less likely to interrupt a quick deployment because it takes less time.

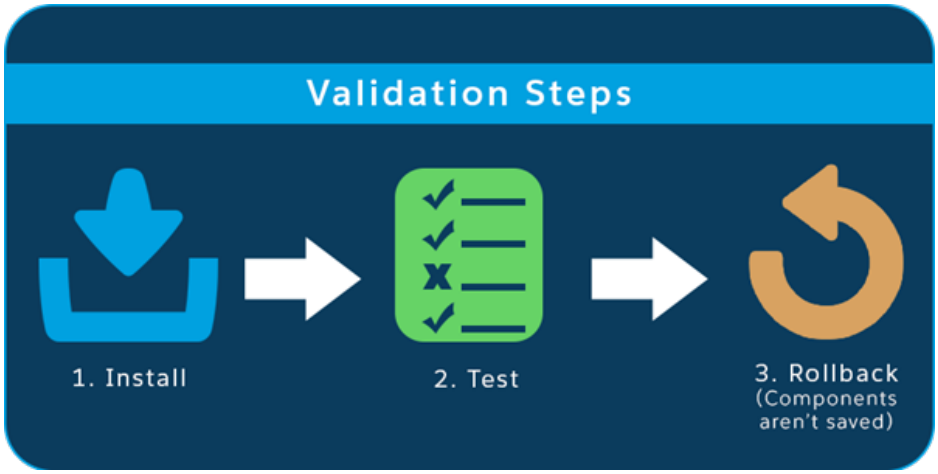
Suppose that your team updates many apps and settings and has developed a complex supply management app. Your team decides to have all local tests in the org run to ensure the quality of the deployment. In this scenario, we recommend that you first run a validation in production to verify the results of your tests and fix test failures. A validation is a deployment that's used only to check the results of deploying components. It doesn't save any components in the org. You can view the success and failure messages that you would receive with an actual deployment. After you fix test failures, and all tests pass as part of the validation, you can run a quick deployment, which skips tests and saves time.

You can run a quick deployment when the following requirements are met.

- The components have been validated successfully for the target environment within the last 10 days.
- As part of the validation, Apex tests in the target org have passed.
- Code coverage requirements are met.
    - If all tests in the org or all local tests are run, overall code coverage is at least 75%, and Apex triggers have some coverage.
    - If specific tests are run with the **Run specified tests** test level, each class and trigger that was deployed is covered by at least 75% individually.

We recommend starting a validation during off-peak usage time and limiting changes to your org while the validation is in progress. The validation process locks the resources that are being deployed. Changes you make to locked resources or items related to those resources while the validation is in progress can result in errors.

To perform a quick deployment, first run a validation (a check-only deployment) with Apex test execution on the set of components to deploy. If your validation succeeds and qualifies for a quick deployment, you can start a quick deployment.



Perform a Quick Deployment with the Force.com Migration Tool

You can perform a quick deployment in version 34.0 (Summer '15) or later of the Force.com Migration Tool. To download the latest version, go to the Force.com Migration Tool page in Salesforce Developers. To learn how to set up the tool, see the Force.com Migration Tool Guide.

The prerequisite for a quick deployment is a recent validation with passing tests. The following Ant target in build.xml includes the `checkOnly=true` parameter to indicate that the deployment is a validation.

```
<target name="deployCodeCheckOnly">
      <sf:deploy username="${sf.username}" password="${sf.password}"
                        sessionId="${sf.sessionId}"
serverurl="${sf.serverurl}"
                        maxPoll="${sf.maxPoll}" deployRoot="codepkg"
                        testLevel="RunLocalTests"  checkOnly="true"/>
</target>
```

Note

The validation target given in this example contains the `testLevel="RunLocalTests"` parameter, which causes local tests to execute. You'll learn more about test levels in the next section.

After you run this validation (for example, by using the command `ant deployCodeCheckOnly`) and verified that the tests passed, you can run a quick deployment. The output of the validation command displays the ID of the validation. Copy this ID, and use it for the quick deployment.

In `build.properties`, replace the following line with the validation ID you copied.

```
sf.recentValidationId = <Deployment ID of the
validation>
```

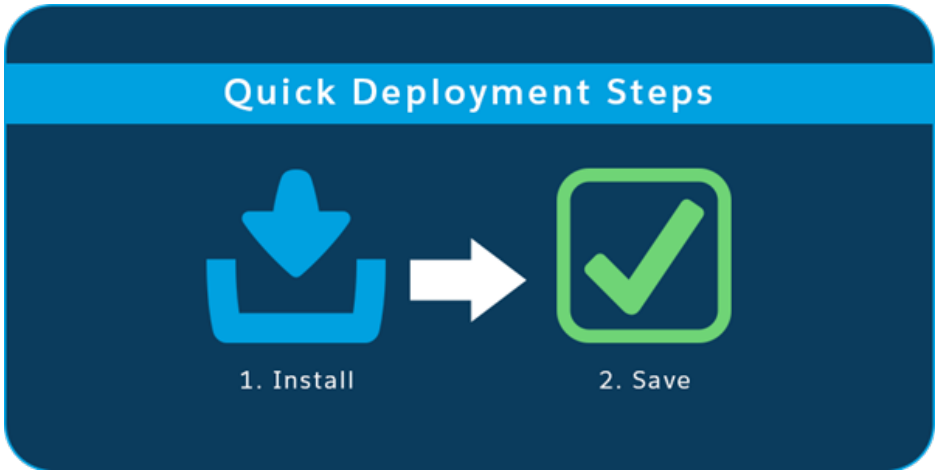
For example:

```
sf.recentValidationId =
0AfD0000000oLp0KAE
```

The following Ant target in `build.xml` performs a quick deployment and references the `sf.recentValidationId` property that you've just set.

```
<target name="quickDeploy">
  <sf:deployRecentValidation username="${sf.username}"
                             password="${sf.password}"
                             sessionId="${sf.sessionId}"
                             serverurl="${sf.serverurl}"
                             maxPoll="${sf.maxPoll}"
                             recentValidationId="${sf.
recentValidationId}"/>
</target>
```

Perform a deployment using this target (for example, `ant quickDeploy`).



Test Levels

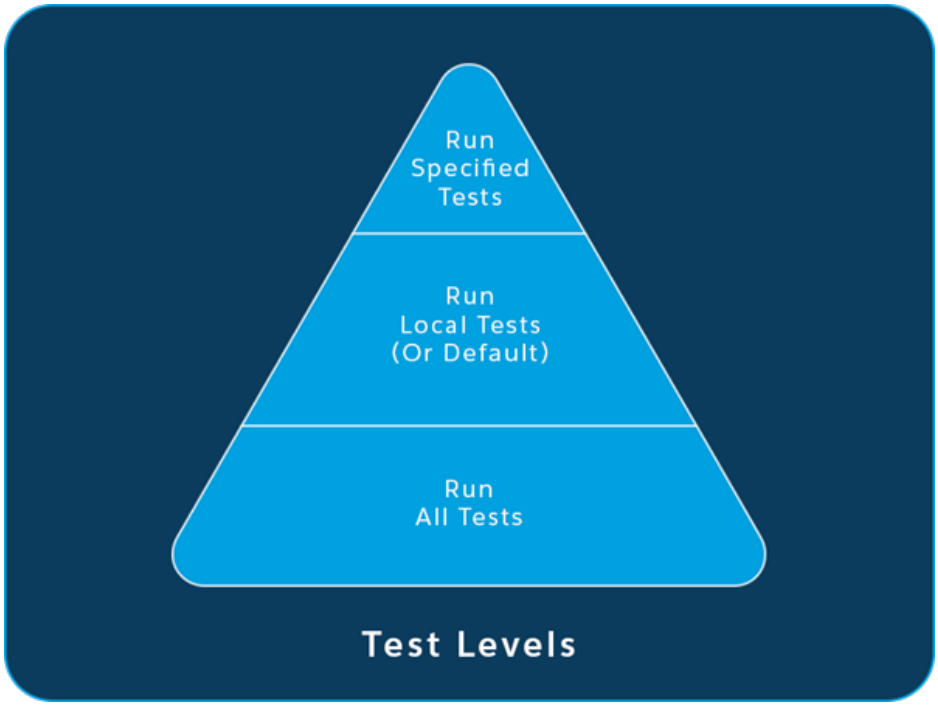
Test levels enable you to choose which tests are run in a validation or deployment. The following test levels are available.

RunSpecifiedTests
: Only the tests that you specify are run. Choose the test classes that provide sufficient coverage for the classes and triggers being deployed. Code coverage requirements differ from the default coverage requirements when using this level in production. The executed tests must cover the deployed class or trigger with a minimum of 75% code coverage. This coverage is computed for each class or trigger individually and is different from the overall coverage percentage.

RunLocalTests
: All tests in your org are run, except the ones that originate from installed managed packages. This test level is the default for production deployments that include Apex classes or triggers.

RunAllTestsInOrg
: All tests in your org are run, including tests of managed packages.

Default (no test level set)
: In production, all local tests are executed if your change set contains Apex classes or triggers. If your package doesn't contain Apex components, no tests are run. In development environments, such as sandbox, Developer Edition, or trial orgs, no tests are run by default.



These are some useful scenarios for test levels.

Run a Subset of Tests

Suppose your development team created an app to display all accounts on a map. The app contains two Apex classes and two test classes. By default, all local tests in the target org are run if you're deploying Apex code. To deploy this app to production, you don't want to run the hundreds or thousands of preexisting test classes in production. Instead, you want to run only the two test classes in your app that provide sufficient test coverage for your app's Apex classes. In this case, you use the RunSpecifiedTests test level to indicate the two test classes to run.

Enable Testing When Not Required by Default

You can explicitly enable testing when it is turned off by default. For example, when you deploy a custom object and no Apex code, tests are turned off by default. But if an Apex class uses this custom object, you might want to run some test classes. In this case, turn on testing when deploying non-Apex components.

In non-production environments, such as sandboxes or Developer Edition orgs, Apex tests aren't run in deployments if a test level isn't set. You can enable testing by setting a test level. That way, you can fix test failures early before deploying to production.

Run a Subset of Tests with the Force.com Migration Tool

You can use test levels in version 34.0 (Summer '15) or later of the Force.com Migration Tool. To download the latest version, go to the Force.com Migration Tool page in Salesforce Developers. To learn how to set up the tool, see the Force.com Migration Tool Guide.

Ant targets for the Migration Tool are specified in the build.xml file. The following target specifies three test classes to run and uses the RunSpecifiedTests test level. Choose tests that provide sufficient code coverage for each Apex class and trigger that you're deploying.

```
<target name="deployCode">
      <!-- Upload the contents of the "codepkg" directory, -->
      <!-- and run the three test classes. -->
      <sf:deploy username="${sf.username}" password="${sf.password}"
                 sessionId="${sf.sessionId}"
serverurl="${sf.serverurl}"
                 maxPoll="${sf.maxPoll}" deployRoot="codepkg"
                 testLevel="RunSpecifiedTests" rollbackOnError="true">
          <runTest>TestClass1</runTest>
          <runTest>TestClass2</runTest>
          <runTest>TestClass3</runTest>
      </sf:deploy>
</target>
```

Note

If your org has a namespace defined, prepend the namespace name to your test class names. For example, `MyNamespace.TestClass1`. In the previous example, the list of test classes would be `MyNamespace.TestClass1`, `MyNamespace.TestClass2`, and `MyNamespace.TestClass3`.

Test Resumption in Deployments

When a Salesforce service update interrupts a deployment, Salesforce resumes the deployment after the service has been restored. To avoid lost time, Salesforce resumes the deployment by running the tests that haven't yet been run. A small number of tests might be rerun if their results weren't saved. Salesforce saves the results of executed tests in chunks. Therefore, a small portion of tests might not have their results saved because they weren't part of the last saved chunk.

The majority of deployment time is spent in test execution, so resuming tests saves you a lot of time. This enhancement has been applied for Metadata API–based deployments starting in Winter '16 and for change sets in Spring '16.

Resources

Use Version Control

Learning Objectives

After completing this unit, you'll be able to:

- Explain when it's important to use version control.
- Retrieve and deploy files to and from version control.
- Implement a continuous integration process.

Version Control

In traditional software development you have a code stack that represents the "source of truth". This source resides in a version control system, and is iteratively developed and succeeded by new versions, or rolled back to previous versions. Sometimes new features are branched, meaning one code line forks off for more isolation from the main code line, and then oftentimes the branches are merged back into to the main line when that feature is ready. For most developers, this is business as usual.

In cloud computing, the source of truth is what's on the production org. Whatever is running on the server is the main line. When you retrieve code or metadata to a version control system, you're receiving a metadata representation of what's on the server. If someone were to change what's on the server (making a change in the production org), then the source of truth has changed, regardless of what's in your repository. This could lead to overwriting the changes on production, and/or a difficulty deploying changes.

In addition, your version control repository only copies the code and metadata types that are exposed in the Metadata API. This means you can create things in a development environment (sandbox, developer org) that can't be stored in source control.

When Is Version Control Not Necessary?

- Small-scale development - If you have only one or two developers, version control may be additional overhead, without much benefit.
- Change sets - If you're using change sets, then you already have cloud-based storage for metadata. It isn't version control, but every time you create an outbound change set, you are saving the metadata from your org at a point in time. You can go back and inspect or revert to those versions any time. Note that as your organization and number of sandboxes grows, you will need version control.

When Should You Use Version Control?

- No sandbox - For organizations that can't use change sets (partners using Developer Edition, or other orgs that don't have a sandbox available), some means of saving metadata is necessary, and this usually means a version control system.
- Regulatory reasons - Some highly regulated industries, such as banking, healthcare, and legal, require version control for regulatory reasons.
- Backup copies - Perhaps you want a copy of your org settings, so that you can revert to older versions if necessary, or populate a new org with the same settings.
- Personal preference - Perhaps you or your company believes that having local copies of files is safer than leaving your metadata on the server.
- Convention - Using version control is considered to be a general best practice for software development and you may already be using a version control system for other development needs.
- Large-scale development - If you have many developers and features, then version control becomes more and more important as people overwrite each other's changes and rollbacks become more common.

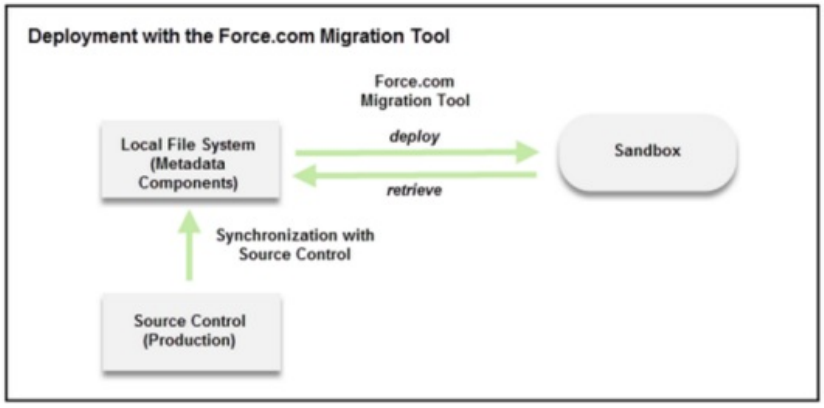
Source Control Systems

A source control system ensures a quality development process by maintaining a separate branch for each project without overwriting changes made as

part of other projects. Team members and developers on other teams can add changes to the source control repository. The repository integrates the changes implemented by concurrent development and separates the different versions of an application.

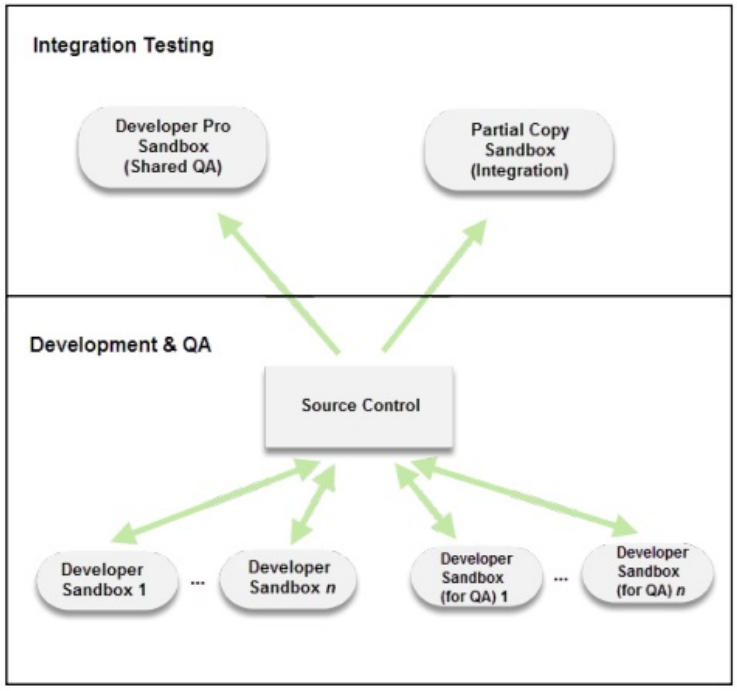
Your repository can contain multiple branches, with each branch containing changes developed by a separate team. For example, one team can work on a feature that's going live in February, while another team works on a feature that's going live in March. These teams need separate developer organizations and integration test sandboxes. You can use a separate branch for a patch release, because it contains bug fixes and different customization metadata and Apex code than the new feature.

The Force.com IDE has integrated source control (such as subclipse) that allows you to instantly see changes you made from the branch and revert if necessary. However, most enterprise development teams use the Force.com Migration Tool to retrieve and deploy between sandbox and source control. Moving the files to a local file system is necessary, as depicted in the following image.



Giving each developer a sandbox provides more control. Each developer can decide when to refresh their sandbox with the latest changes from the repository or when to commit changes.

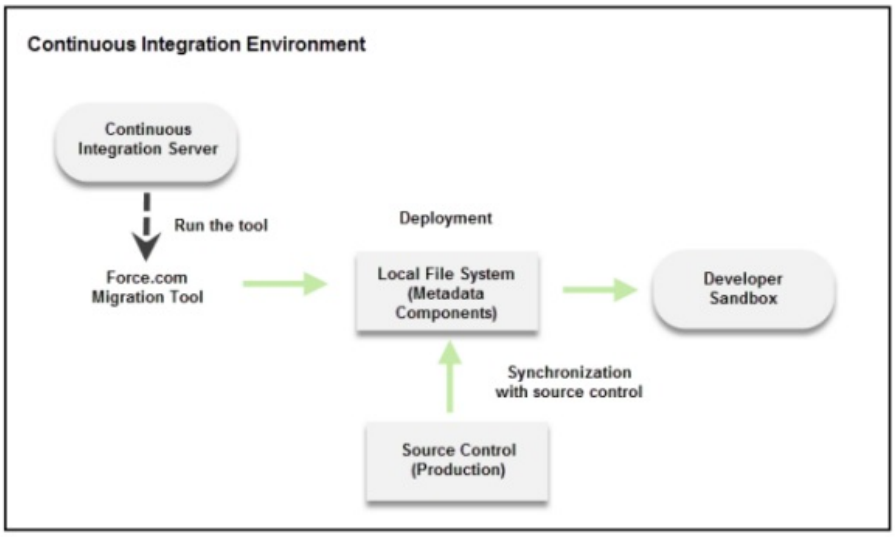
The following diagram illustrates a layout of sandbox environments for development and testing.



Continuous Integration System

For a robust quality assurance process, use a continuous integration system to run test deployments and Apex tests for each change to the source control repository. You can perform these deployments in a dedicated sandbox for continuous integrations.

You can use a continuous integration system to automate deployments. For example, you can use Jenkins to automate deployments to the user acceptance testing (UAT) sandbox. The following diagram shows the relationship between the continuous integration server, sandbox, and source control.



Enterprise development with source control is a complex topic. It requires experimentation and learning to refine the process. For more information, see the resources listed in this unit, especially the Advanced Development Lifecycle Scenario in the Development Lifecycle Guide.

To configure the Force.com Migration Tool to enable tests, see Tips for Configuring the Force.com Migration Tool to Run Tests.

Resources

Get Started with Platform Cache

Learning Objectives

After completing this unit, you'll be able to:

- Describe what platform cache is and what it's used for.
- List the two types of platform cache and give examples of each.
- Describe partitions and how to use them.

What Is a Cache?

A cache is temporary storage. In the computer world, cache is temporary storage for frequently accessed data from a database. Here's an analogy. Suppose you're a chipmunk looking for nuts and acorns for dinner. It's 5:00 and you're ready to eat. Are you going to use the nuts and acorns stored in your cheeks (cache), or are you going back to the forest to gather more from trees (database)? If you access the temporary cache of food in your cheeks, your dinner is closer and you get to eat it faster! Also, you accomplish your goal more efficiently. A data cache has similar advantages, but for people, not chipmunks.

What Is Platform Cache?

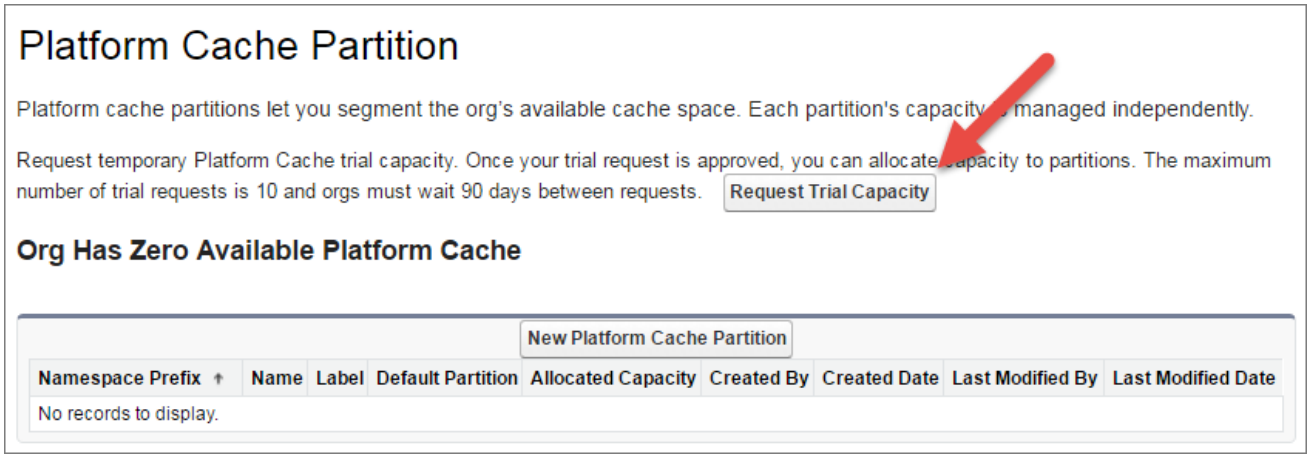
Platform Cache is a memory layer that stores Salesforce session and org data for later access. When you use Platform Cache, your applications can run faster because they store reusable data in memory. Applications can quickly access this data; they don't need to duplicate calculations and requests to the database on subsequent transactions. In short, think of Platform Cache as RAM for your cloud application.

With Platform Cache, you can also allocate cache space so that some apps or operations don't steal capacity from others. You use partitions to distribute space. We'll get to partitions later.

Before We Go Any Further

Let's pause for a moment for you to request a trial of Platform Cache. By default, your Developer org has 0 MB cache capacity. You can request a trial cache of 10 MB.

To request a trial, go to Setup in your Developer org. In the Quick Find box, enter `cache`, and then click **Platform Cache**. Click **Request Trial Capacity** and wait for the email notifying you that your Platform Cache trial is active. Salesforce approves trial requests immediately, but it can take a few minutes for you to receive the email.



If you don't have a cache trial, you can still execute cache operations to learn how to use the cache. However, cache storage is bypassed, and retrieved values are null (cache misses).

Okay, now that you've requested a Platform Cache trial, let's learn some more concepts.

When Can I Use Platform Cache?

You can use Platform Cache in your code almost anywhere you access the same data over and over. Using cached data improves the performance of your app and is faster than performing SOQL queries repetitively, making multiple API calls, or computing complex calculations.

The best data to cache is:

- Reused throughout a session, or reused across all users and requests
- Static (not rapidly changing)

- Expensive to compute or retrieve

Store Data That Doesn't Change Often

Use the cache to store static data or data that doesn't change often. This data is initially retrieved through API calls from a third party or locally through SOQL queries. If the data changes, cache this data if the values don't have to be highly accurate at all times.

Examples of such static data are:

- Public transit schedule
- Company shuttle bus schedule
- Tab headers that all users see
- A static navigation bar that appears on every page of your app
- A user's shopping cart that you want to persist during a session
- Daily snapshots of exchange rates (rates fluctuate during a day)

Store Data Obtained from Complex Calculations

Values that are a result of complex calculations or long queries are good candidates for cache storage. Examples of such data are:

- Total sales over the past week
- Total volunteering hours company employees did as a whole
- Top sales ranking

For clues on where to use Platform Cache, inspect your code. Do you currently store app data by using custom objects and custom settings, or by overloading a Visualforce view state? These storage values are all candidates for Platform Cache.

Not every use case is a Platform Cache use case. For example, data that changes often and that is real-time, such as stock quotes, isn't a good candidate for caching. Also, ensure you familiarize yourself with Platform Cache limitations. For example, if your data is accessed by asynchronous Apex, it can't be stored in a cache that is based on the user's session.

Cache Allocations by Edition

Platform Cache is available to customers with Enterprise Edition orgs and above. The following editions come with some default cache space, but often, adding more cache gives even greater performance enhancements.

- Enterprise Edition (10 MB by default)
- Unlimited Edition (30 MB by default)
- Performance Edition (30 MB by default)

Experiment with Trial Cache

You can purchase additional cache for your org. To determine how much extra cache would be beneficial for your applications, you can request trial cache and try it out. Also, request trial cache for Professional Edition before purchasing cache. Use trial cache in your Developer Edition org to develop and test your applications with Platform Cache. When your request is approved, you receive 30 MB of trial cache space (10 MB for Developer Edition). If you need more trial cache space, contact Salesforce.

What Are the Types of Platform Cache?

There are two types of Platform Cache: org cache and session cache.

Org Cache

Org cache stores org-wide data that anyone in the org can use. Org cache is accessible across sessions, requests, and org users and profiles.

For example, weather data can be cached and displayed for contacts based on their location. Or daily snapshots of currency exchange rates can be cached for use in an app.

Session Cache

Session cache stores data for an individual user and is tied to that user's session. The maximum life of a session is 8 hours.

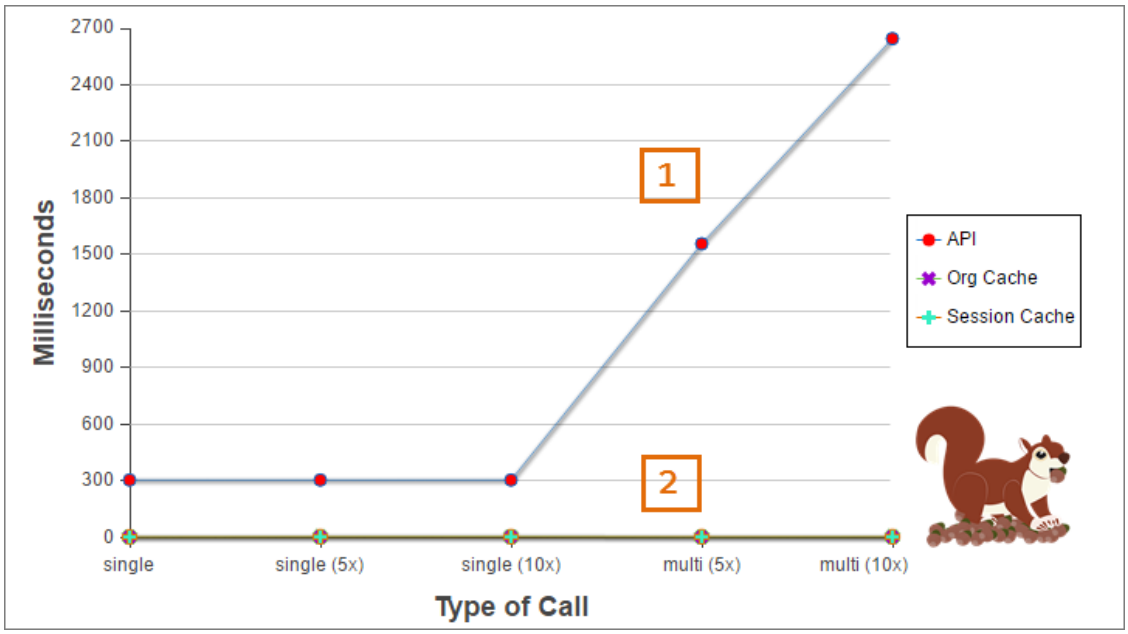
For example, suppose that your app calculates the distance from a user's location to all customers the user wishes to visit on the same day. The location and the calculated distances can be stored in the session cache. That way, if the user wants to get this information again, the distances don't need to be recalculated. Or, you might have an app that enables users to customize their navigation tab order and reuse that order as they visit other pages in the app.

What Are the Performance Gains When Using the Cache?

You might be wondering how much performance your app gains by using Platform Cache. Retrieving data from the cache is much faster than through an API call. When comparing SOQL to cache retrieval times, the cache is also much faster than SOQL queries.

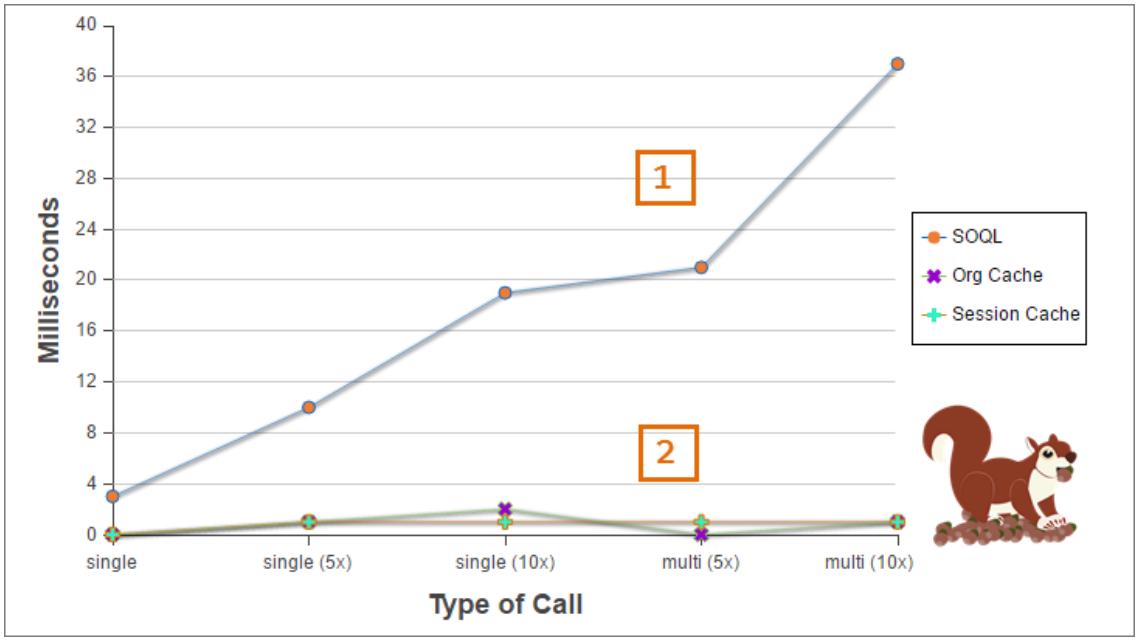
The following chart shows the retrieval times, in milliseconds, of data through an API call and the cache. It is easy to notice the huge performance gain when fetching data locally through the cache, especially when retrieving data in multiple transactions. In the sample used for the graph, the cache is hundreds of times faster than API calls. In this graph, cache retrieval times are just a few milliseconds but appear as almost zero due to the scale used for the time value. Keep in mind that this chart is a sample test, and actual numbers might vary for other apps.

Making API Calls to External Services Is Slower (1) Than Getting Data from the Cache (2).



This next graph compares SOQL with org and session cache retrieval times. As you can see, SOQL is slower than the cache. In this example, the cache is two or more times faster than SOQL for data retrieval in a single transaction. When performing retrievals in multiple transactions, the difference is even larger. (Note that this graph is a sample and actual numbers might vary for other apps.)

Getting Data Through SOQL Queries (1) Is Slower Than Getting Data from the Org and Session Cache (2).



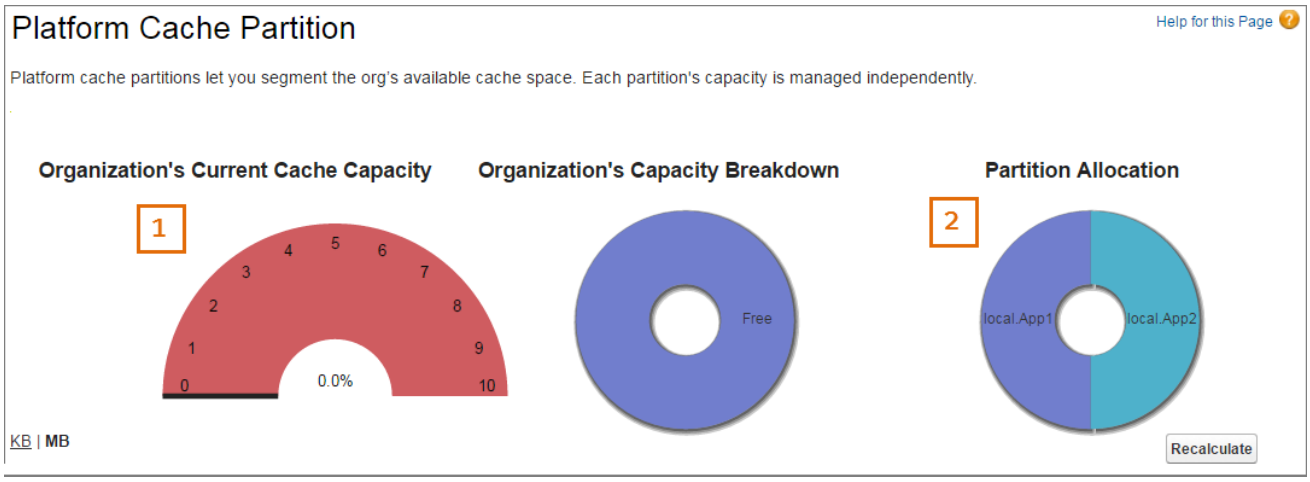
What Are Cache Partitions?

Remember when we mentioned earlier that with Platform Cache you can allocate space using partitions? Let's talk now about partitions. Partitions let you allocate cache space to balance usage and performance across apps. Caching data to designated partitions ensures that the cache space isn't

overwritten by other apps or by less critical data.

Before you can use cache space in your org, you must create partitions to define how much capacity you want for your apps. Each partition capacity is broken down between org cache and session cache. Session and org cache allocations can be zero, 5 MB, or greater, and must be whole numbers. The minimum size of a partition, including its org and session cache allocations, is 5 MB. For example, say your org has a total of 10-MB cache space and you created a partition with a total of 5 MB, 5 MB of which is for session cache and 0 MB for org cache. Or you can create a partition of 10-MB space, with 5 MB for org cache and 5 MB for session cache. The sum of all partitions, including the default partition, equals the Platform Cache total allocation.

The following image shows charts of cache capacity and partition allocation. In this example, we haven't used the cache yet as evidenced by the 0% cache usage (1) and two partitions have been created with equal allocations (2).



Default Partition

You can define any partition as the default partition, but you can have only one default partition. The default partition enables you to use shorthand syntax to perform cache operations on that partition. This means that you don't have to fully qualify the key name with the namespace and partition name when adding a key-value pair. For example, instead of calling `Cache.Org.put('namespace.partition.key', 0);` you can just call `Cache.Org.put('key', 0);`

In the next unit, you'll create a partition in Setup to get started using Platform Cache!

Use Org & Session Cache

Learning Objectives

After completing this unit, you'll be able to:

- Create a partition.
- Store and retrieve values in the org and session cache.
- Describe how long values last in the cache.
- Handle cache misses.
- Read session cache from a Visualforce page.

Create a Partition

Note

Don't have a cache trial yet? This unit requires an active Platform Cache trial. Request a trial using the instructions in the previous unit. If you don't have a cache trial, you can still carry out the steps in this unit, but your data won't be found in the cache.

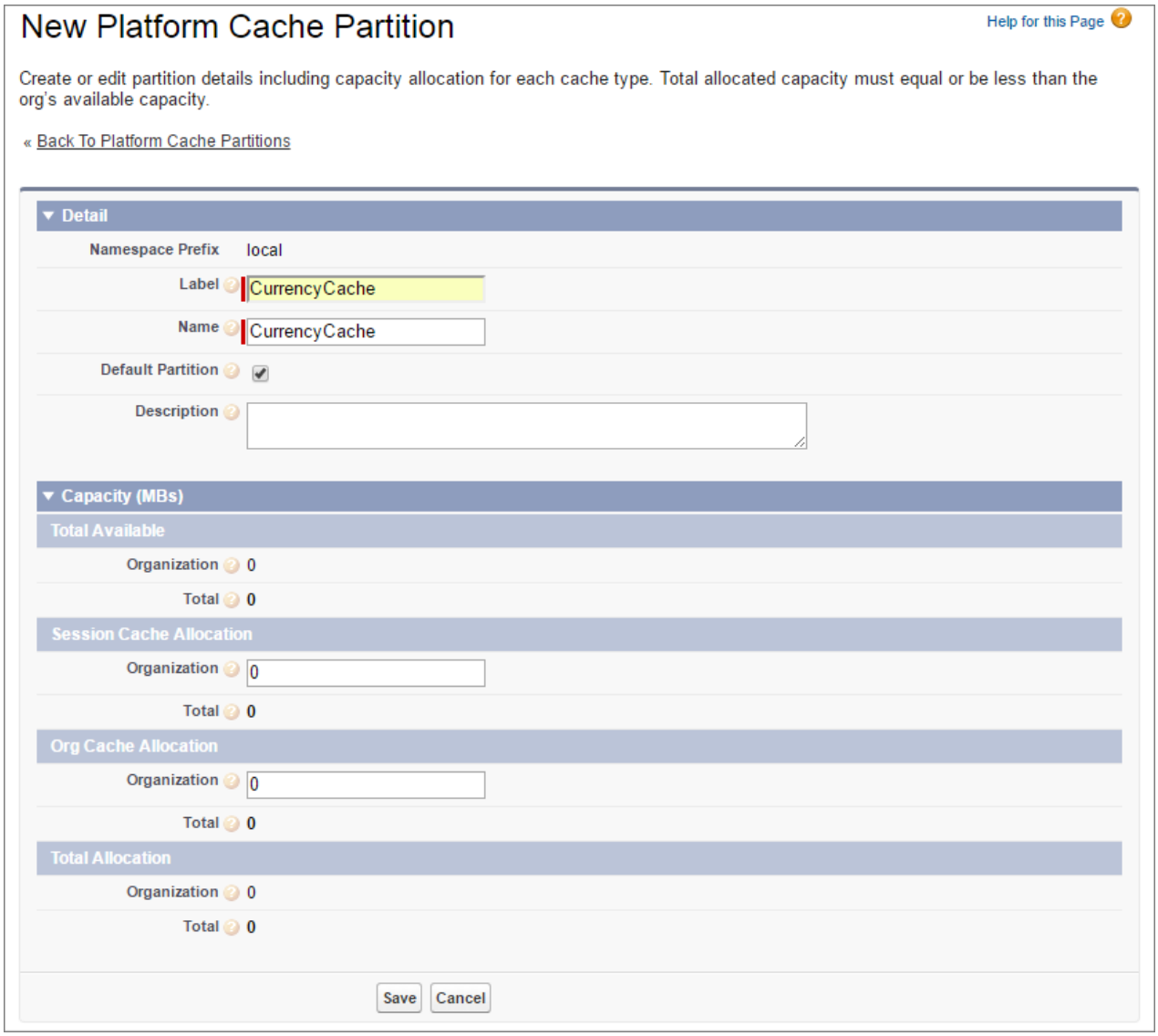
To use Platform Cache, first set up at least one partition. Once you've set up partitions, you can add, access, and remove data from them using the Platform Cache Apex API.

Each partition has one session cache and one org cache segment. You can allocate separate capacity to each segment. Usually, you allocate at least 5 MB to your partition. For this example, we aren't allocating any space to ensure that your code correctly handles cache misses. When no space is allocated, the cache miss rate is 100%, which means that cache values aren't found in the cache, and the `get()` method returns `null`. Consider using

this technique to test cache misses. We cover how to handle cache misses in a later section.

To get started, let's create one partition from the Platform Cache page (available in Salesforce Classic only).

1. In Setup, enter `Platform Cache` in the Quick Find box, then select **Platform Cache**.
2. Click **New Platform Cache Partition**.
3. Give the partition a name (such as the name of your application).
4. Check **Default Partition**.
5. Enter `0` for session cache and `0` for org cache, and then click **Save**.



Cache Key Name Format

Each cache key has the following format:

```
Namespace.Partition.Key
```

Namespace is the namespace name of the org where the app is running, which can also be set to the special name "local". The "local" name refers to the namespace of your org whether a namespace is defined in the org or not.

Partition is the name of the partition you created. In this example, it is CurrencyCache.

Key is the name of the key you used to store a value. The key name uniquely represents your cached value.

Let's say we'd like to store the currency exchange rate from US Dollar to Euro. We can create a key named DollarToEuroRate. For the partition we've just created, the full name of the key is:

```
local.CurrencyCache.DollarToEuroRate
```

For example, this snippet stores a value in the org cache for the DollarToEuroRate key.

```
Cache.Org.put('local.CurrencyCache.DollarToEuroRate',
'0.91');
```

The partition we created is a default partition, so you can omit the namespace and partition name and just specify the key name.

```
DollarToEuroRate
```

When using a default partition, you can shorten the put() call to the following.

```
Cache.Org.put('DollarToEuroRate',
'0.91');
```

Store and Retrieve Data in Org Cache

You've finished setting up partitions, which is the only step you do in the user interface. Now we'll switch to Apex to manage the cache. Use org cache to store data that is available to anyone in the org. Either use the `Cache.Org` class methods, or use the `Cache.OrgPartition` class to reference a specific partition. Then call the cache methods on that partition.

The following Apex snippet shows how to access a partition using the `Cache.OrgPartition` class to store and retrieve a cache value for a currency exchange application. It obtains the partition named CurrencyCache in the local namespace. A new value is added with key DollarToEuroRate and today's currency exchange rate. Next, the value for key DollarToEuroRate is retrieved from the cache.

```
// Get partition
Cache.OrgPartition orgPart =
Cache.Org.getPartition('local.CurrencyCache');

// Add cache value to the partition. Usually, the value is obtained from a
// callout, but hardcoding it in this example for simplicity.
orgPart.put('DollarToEuroRate', '0.91');

// Retrieve cache value from the partition
String cachedRate = (String)orgPart.get('DollarToEuroRate');
```

If you're managing cache values in just one partition, use the `Cache.OrgPartition` methods. The `Cache.OrgPartition` methods are easier to use than the `Cache.Org` methods because you specify the namespace and partition prefix only once when you create the partition object.

Do Cached Values Last Forever?

The previous example assumes that everything goes correctly, namely that:

- Platform Cache is enabled and has a partition with available space
- The cached value is found in the cache
- The value is successfully stored in the cache

But in real life, cached data is not always guaranteed. Platform Cache is intended as a temporary space. For example, the cache might have expired. Even if the cache is still alive, it is possible that your cached data might be evicted from the cache. Just like chipmunks clean out their cheeks to make space for more acorns, Platform Cache also clears some space for more data! When the partition limit is exceeded, Salesforce evicts cached data based on a least recently used (LRU) algorithm. The cache eviction takes place until usage is reduced to less than or equal to 100% capacity. Also, if you exceed the local cache limit, items can be evicted from the local cache before the request has been committed.

Cached Data Duration and Expiration

The amount of time during which data is kept in the cache is called the time-to-live value (ttlsecs). You specify the time-to-live value when you store a key-value pair in the cache using Apex methods. For session cache, your data can live up to 8 hours in the cache. For org cache, your data can live up to 48 hours in the cache. By default, the time-to-live value for org cache is 24 hours.

Session cache expires when its specified time-to-live value is reached or when the user session expires, whichever comes first. Org cache expires when its specified time-to-live value is reached.

Best Practice for Handling Cache Misses

As a best practice, your code should anticipate and accommodate points of failure. In other words, always assume that a cache miss can happen. A cache miss is when you request a value for a key from the cache but the value is not found. The value that your `get()` call returns is null. Check the result of the `get()` call for null and handle it accordingly. For example, if the result is `null`, get the value from the database or an API call.

The following Apex snippet shows how to handle a cache miss by checking whether the returned value from the cache is not `null` (`if (cachedRate != null)`). If the value is not `null`, you can use the value, for example, to display it on a page. Otherwise, fetch this value from another source, such as an API call or from Salesforce.

```
Cache.OrgPartition orgPart =
Cache.Org.getPartition('local.CurrencyCache');
String cachedRate = (String)orgPart.get('DollarToEuroRate');
// Check the cache value that the get() call returned.
if (cachedRate != null) {
    // Display this exchange rate
} else {
    // We have a cache miss, so fetch the value from the source.
    // Call an API to get the exchange rate.
}
```

Store and Retrieve Data in Session Cache

Remember what session cache does? That's right, it stores data that is tied to individual user sessions. For example, you might use session cache in an app to store the user's favorite currency or a user's custom navigation tab order. With session cache, you can manage cache values in Apex and read cached values with a Visualforce global variable.

When using Apex, managing the session cache is similar to the way you manage the org cache, except the class names are different. Use the `Cache.Session` and `Cache.SessionPartition` classes to access values stored in the session cache. To manage values in any partition, use the methods in the `Cache.Session` class. If you're managing cache values in only one partition, use the `Cache.SessionPartition` methods instead. The `Cache.SessionPartition` methods are easier to use than the `Cache.Session` methods because you specify the namespace and partition prefix only once when you create the partition object.

The following snippet of Apex code shows how to access a partition to store and retrieve a cache value. It obtains the partition named CurrencyCache in the local namespace. A new value is added with key FavoriteCurrency. The value for the FavoriteCurrency key is retrieved. The FavoriteCurrency key stores the user's favorite currency, so this value is different for each user and hence a good candidate for session cache.

```
// Get partition
Cache.SessionPartition sessionPart =
Cache.Session.getPartition('local.CurrencyCache');

// Add cache value to the partition
sessionPart.put('FavoriteCurrency', 'JPY');

// Retrieve cache value from the partition
String cachedRate = (String)sessionPart.get('FavoriteCurrency');
```

Note

Session cache doesn't support Anonymous Apex blocks. For example, if you execute the previous snippet of session cache methods in the Developer Console, you'll get an error. However, you can execute org cache methods through Anonymous Apex.

Access Session Cache with Visualforce Global Variables

Access cached values stored in the session cache from a Visualforce page by using the `$Cache.Session` global variable. By using this global variable, you can read cached values that were stored with Apex directly from your Visualforce page.

When using the `$Cache.Session` global variable, fully qualify the key name with the namespace and partition name. This example is an output text component that retrieves a cached value from the namespace `ExPro`, partition `CurrencyCache`, and key `FavoriteCurrencyRate`.

```
<apex:outputText value="
{!$Cache.Session.ExPro.CurrencyCache.FavoriteCurrencyRate}"/>
```

Note

The Visualforce global variable is available only for session cache and not for org cache.

Unlike with Apex methods, you can't omit the `namespace.partition` prefix to reference the default partition in the org. If a namespace isn't defined for the org, use `local` to refer to the namespace of the current org where the code is running.

```
<apex:outputText value="
{!$Cache.Session.local.MyPartition.Key}"/>
```

If the cached value is a data structure that has properties or methods, like an Apex List or a custom class, access those properties in the `$Cache.Session` expression using dot notation. For example, this markup invokes the `List.size()` Apex method if the value of `numbersList` is declared as a List.

```
<apex:outputText value="
{!$Cache.Session.local.MyPartition.numbersList.size}"/>
```

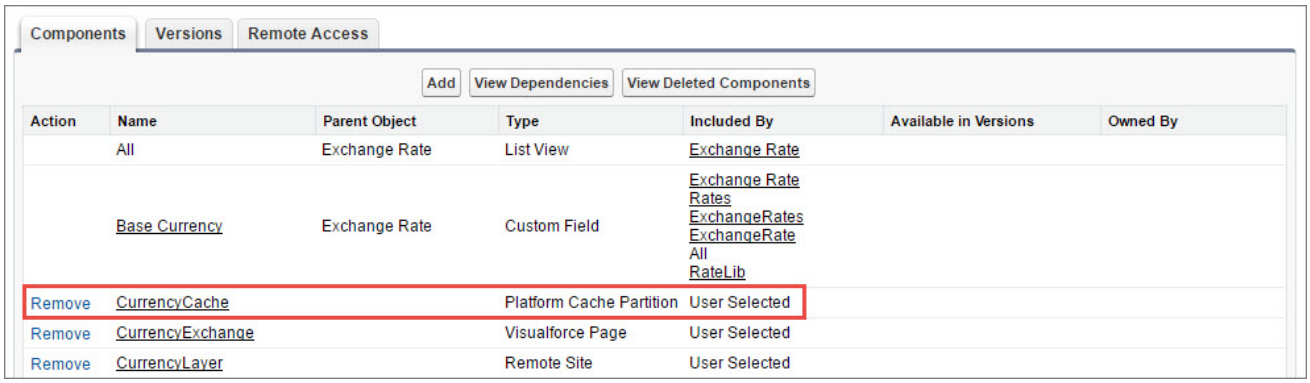
This example accesses the value property on the `myData` cache value that is declared as a custom class.

```
<apex:outputText value="
{!$Cache.Session.local.MyPartition.myData.value}"/>
```

Protected Cache Allocation for ISV Apps

By using platform cache, ISV apps run faster and have better performance. If you're an ISV developer, you can guarantee cache space for your apps by purchasing cache space for your own namespace. That way, when your app is installed in a subscriber org, the space for your app's cache is not affected by the usage of the cache in the subscriber's org. Only Apex code running from your app's namespace can access and use your namespace's cache. No other code in the subscriber org can use this cache. You can test your app against your namespace's cache and be assured that the cache allocation is going to be protected in every subscriber org.

Cache partitions are distributed to subscribers as part of the app's package. Add one or more cache partitions for your namespace to your package as components in the same way as you add other components. Cache partitions aren't automatically added as dependent components.

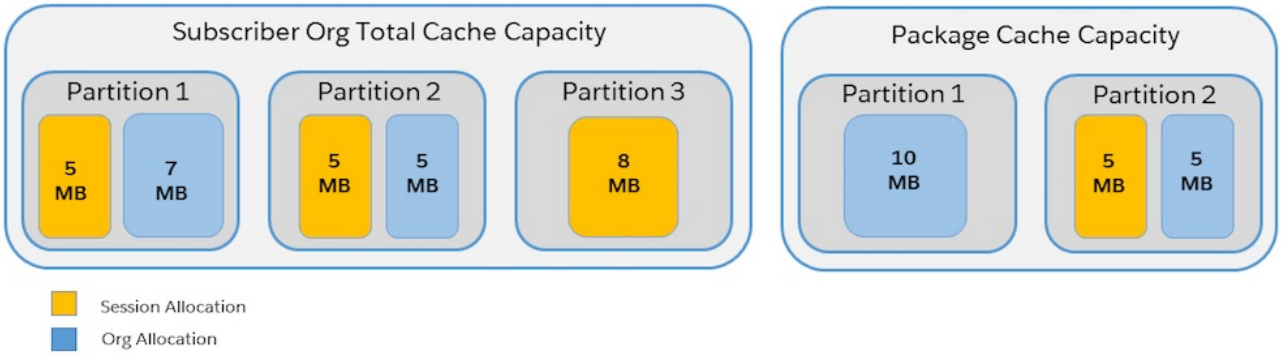


Note

See the following about packaged partitions.

- The partitions you add to the package must be nondefault partitions.
- When a subscriber installs your package, the cache partitions get installed in their org enabling your app to access the installed cache partitions for your namespace.
- Subscribers must have an Enterprise Edition or Unlimited Edition org, or must have purchased Platform Cache.

The following diagram shows an example cache capacity for an Enterprise Edition subscriber org in which a package was installed with packaged cache. The subscriber's total cache capacity of 30 MB is divided into three partitions consisting of combinations of session and org cache. The packaged cache capacity originates from the installed package in the subscriber's org and contains two partitions. The packaged cache consists of 20 MB of cache that the ISV purchased. (Remember, only Apex code from the package can access the packaged cache and code in the subscriber's org can't use this cache.)



Purchase your namespace cache from the Channel Order App. You can purchase cache in 10-MB blocks. To determine how much cache space your apps need, test your apps with your cache partitions. Use trial cache to increase the capacity of cache in your org. As we mentioned in the first unit, certain editions of subscriber orgs get cache allocation by default. For your Developer Edition org in which you develop apps, you can request 10 MB of trial cache. You can request to increase the amount of trial cache given to your org by contacting Salesforce. By experimenting with different cache sizes, you'll have a better idea of how much cache to purchase for your own namespace.

Now that you've seen how to get and store values in Platform Cache, try it on your own! Complete the following challenge to test your knowledge.

Resources

Walk Through a Sample Application and Discover Cache Diagnostics

Learning Objectives

After completing this unit, you'll be able to:

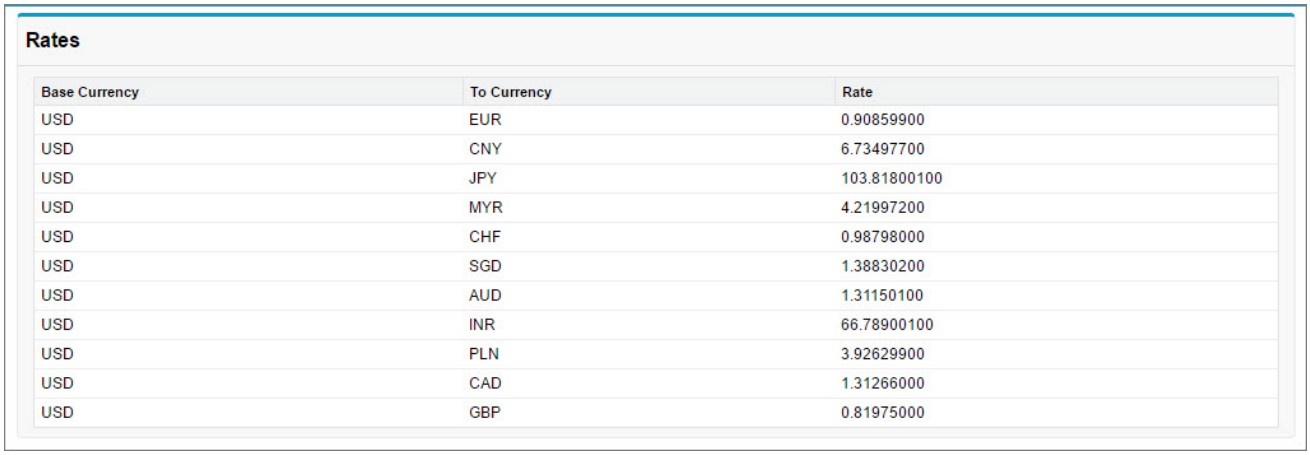
- Explain a pattern for storing and refreshing cached data.
- Decide on the data structure to use for the cached value.
- Diagnose your cache usage.

Sample Application Walkthrough

Let's take a look at a sample application that demonstrates how to use the org cache for storing and retrieving currency exchange rates. Exchange rates fluctuate during a day so this sample app doesn't return real-time rates. The app just provides a daily snapshot of exchange rates. Because we're not interested in accurate real-time values but only in daily values, caching the currency exchange rates is a good choice. When exchange rates are retrieved repeatedly, getting them from the cache saves a lot of time and improves the performance of the app.

Sample Application Overview

We based our exchange rates sample application on a Visualforce page and an Apex controller that contains the logic for fetching exchange rates. The first time this page is accessed, the rates are obtained through an API call to an external web service. Subsequent executions of this page return the rates from the cache as long as the rates are less than a day old. For each exchange rate, this page displays the base currency, the target currency for conversion, and the conversion rate. For illustration purposes, a small set of currencies is returned.



Rates

Base Currency	To Currency	Rate
USD	EUR	0.90859900
USD	CNY	6.73497700
USD	JPY	103.81800100
USD	MYR	4.21997200
USD	CHF	0.98798000
USD	SGD	1.38830200
USD	AUD	1.31150100
USD	INR	66.78900100
USD	PLN	3.92629900
USD	CAD	1.31266000
USD	GBP	0.81975000

This example is the markup of the Visualforce page. This page is associated with the `Rates` Apex controller.

```
<apex:page controller="ExchangeRates" action="{!init}">

   <apex:pageBlock title="Rates">
      <apex:pageBlockTable value="{!Rates}" var="rate">
         <apex:column value="
{!rate.Base_Currency__c}"/>
         <apex:column value="{!rate.To_Currency__c}"/>
         <apex:column value="{!rate.Rate__c }"/>
      </apex:pageBlockTable>
   </apex:pageBlock>

</apex:page>
```

Sample Apex Controller

Our sample Apex controller does the heavy lifting. It fetches exchange rates, stores them in Salesforce and in the cache, and retrieves rates from the cache. Here is a breakdown of the operations that the sample controller does, followed by the source code.

The first time the sample is run, the following operations occur.

- The exchange rates are obtained from an API call to an external endpoint.
- The result (in JSON format) returned from the API call is parsed and saved in Exchange_Rate__c sObjects in Salesforce.
- The `getCachedRates()` method stores the array of Exchange_Rate__c sObjects in the org cache.

On subsequent executions of the sample:

- The sample checks how fresh the stored data is. To this end, it performs a SOQL query to fetch the createdDate value of the first returned Exchange_Rate__c record.
- If the date is older than one day, the exchange rates are obtained from the API call, as in the first execution.
- If the date is newer than one day, the rates are taken from the org cache. If there is a cache miss, the rates are queried from Exchange_Rate__c sObjects and stored in the org cache.

Note

The Apex controller uses a helper Apex class called `RateLib`, which is not shown here. This helper class contains methods to make the outgoing API call to an exchange rate service, parse the JSON result from the API call, and store Exchange_Rate__c records.

```
public class ExchangeRates {
    private String currencies = 'EUR,GBP,CAD,PLN,INR,AUD,SGD,CHF,MYR,JPY,CNY';
    public String getCurrencies() { return currencies;}
    public Exchange_Rate__c[] rates {get; set;}

    //
    // Checks if the data is old and gets new data from an external web service
    // through a callout. Calls getCachedRates() to manage the cache.
    //
    public void init() {
        // Let's query the latest data from Salesforce
        Exchange_Rate__c[] latestRecords = ([SELECT CreatedDate FROM
Exchange_Rate__c
                        WHERE Base_Currency__c =:RateLib.baseCurrencies
                              AND forList__c = true
                        ORDER BY CreatedDate DESC
                        LIMIT 1]);

        // If what we have in Salesforce is old, get fresh data from the API
        if ( latestRecords == null
            || latestRecords.size() == 0
            || latestRecords[0].CreatedDate.date() < Datetime.now().date()) {
            // Do API request and parse value out
            String tempString = RateLib.getLoadRate(currencies);
            Map<String, String> apiStrings = RateLib.getParseValues(
                tempString, currencies);

            // Let's store the data in Salesforce
            RateLib.saveRates(apiStrings);

            // Remove the cache key so it gets refreshed in getCachedRates()
            Cache.Org.remove('Rates');
        }
        // Call method to manage the cache
        rates = getCachedRates();
    }

    //
    // Main method for managing the org cache.
    // - Returns exchange rates (Rates key) from the org cache.
    // - Checks for a cache miss.
    // - If there is a cache miss, returns exchange rates from Salesforce
    //    through a SOQL query, and updates the cached value.
    //
    public Exchange_Rate__c[] getCachedRates() {
        // Get the cached value for key named Rates
        Exchange_Rate__c[] rates = (Exchange_Rate__c[])Cache.Org.get(
            RateLib.cacheName+'Rates');

        // Is it a cache miss?
        if(rates == null) {
            // There was a cache miss so get the data via SOQL
            rates = [SELECT Id, Base_Currency__c, To_Currency__c, Rate__c
                        FROM Exchange_Rate__c
                        WHERE Base_Currency__c =:RateLib.baseCurrencies
                              AND forList__c = true
                              AND CreatedDate = TODAY];
            // Reload the cache
            Cache.Org.put(RateLib.cacheName+'Rates', rates);
        }
        return rates;
    }
}
```

To download the source of the Exchange Rates sample and play with it in your Developer org, see the Resources section.

Best Practices for Cache Management

Pattern for Cache Storage

The `ExchangeRates` Apex class contains methods that encapsulate the logic of initializing and refreshing the cache. If the data is stale or is not found, the `init()` method retrieves new exchange rates through an API call, and then stores them in Salesforce. The `getCachedRates()` method manages

the cache internally. If the cached value is not found, this method retrieves an array of rates from Salesforce and stores it in the cache.

Because our app uses external data, it fetches data from a web service through API calls. It also stores the data as Salesforce records as a backup for refreshing the cache. Apps that don't use external data retrieve Salesforce records by using SOQL and cache it. In this case, the process of cache management is simpler and the cache method implementation is shorter. For example, if your app just uses local data from SOQL, you wouldn't need the `init()` method but only the `getCachedRates()` method.

We recommend that you include all the logic for managing the cache in one method. That way, the cache is manipulated in only one place in your app. The central management of the cache reduces the chance of errors from accessing invalid cache (cache misses) or accidentally overwriting cached values.

Deciding What to Cache

This sample stores an array of sObjects in the cache. Is this approach the best choice of data structure to store? There are tradeoffs with every choice you make. Storing smaller pieces of data, like field values instead of entire sObjects or sObject arrays, reduces your cache usage size. But if you store less data in each key, you might need complex logic to rebuild the data and sObjects, which requires more processing time. Also, caching smaller items instead of a list of items worsens cache performance because of the overhead of serialization and cache commit time. For example, instead of storing a list of rates (referenced by the rates variable in this snippet):

```
Cache.Org.put('Rates', rates);
```

You can store the individual rates, each as a field with its own cache key, as follows.

```
Cache.Org.put('DollarToEuroRate', rateEUR);
Cache.Org.put('DollarToChineseYuan', rateCNY);
Cache.Org.put('DollarToJapaneseYen', rateJPY);
// etc.
```

The decision of what data structure to cache depends on what your app does with the data. For example, if the app converts currencies for the same base currency, store at least the exchange rate for each target currency. You could store each currency rate as an individual key. But to make it easier to display the rates on a Visualforce page, cache the data as a list of sObjects. sObjects increase storage space because they contain system fields, such as the created date, but they reduce logic processing time and increase app and cache performance. Remember that when using the cache, the main goal is to reduce the app's execution time.

Diagnose Cache Usage

So, you've done all this work to implement Platform Cache. How do you know if you're making the best use of the cache? There are a couple ways you can check performance data. One way is to view the diagnostics information in Setup (available in Salesforce Classic only).

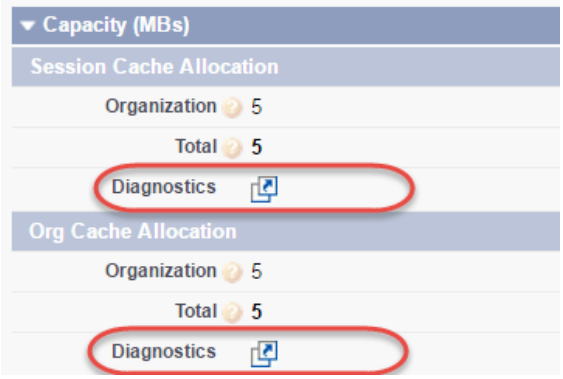
Before you access the diagnostics page, enable the cache diagnostics permission for your user.

1. From Setup, enter `users` in the Quick Find box, then select **Users**.
2. Click your user's name, and then click **Edit**.
3. Select **Cache Diagnostics**, and then click **Save**.

Next, access the diagnostics page for a specific cache type in a partition.

1. From Setup, enter `cache` in the Quick Find box, then select **Platform Cache**.
2. Click the partition for which you want to check diagnostics information.
3. Under session cache or org cache, click **Diagnostics**.

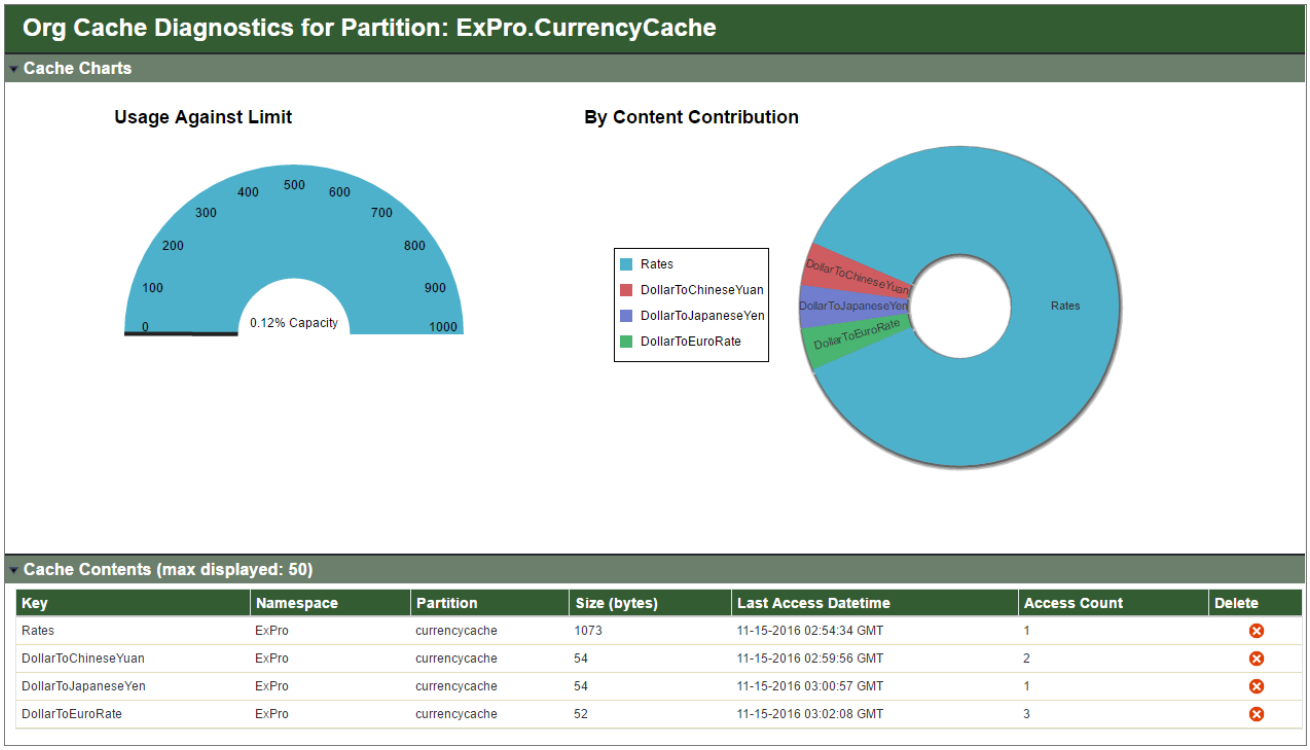
If you click **Diagnostics** for the org cache, the Org Cache Diagnostics page opens in a new tab. This page shows two charts. The first chart (Usage Against Limit) shows your cache usage limit. In our case, we're way below the limit at 0.12%. The second chart (By Content Distribution) is a donut chart that shows cache distribution by key.



Note

Because our sample uses the org cache, we're inspecting the diagnostics page for only the org cache. A similar diagnostics page is also provided for the session cache for apps that use the session cache.

The following image shows a content distribution chart with three cache keys. One of the cache keys, Rates, is used in our exchange rates sample. The Rates key uses up the most space.



Key	Namespace	Partition	Size (bytes)	Last Access Datetime	Access Count	Delete
Rates	ExPro	currencycache	1073	11-15-2016 02:54:34 GMT	1	
DollarToChineseYuan	ExPro	currencycache	54	11-15-2016 02:59:56 GMT	2	
DollarToJapaneseYen	ExPro	currencycache	54	11-15-2016 03:00:57 GMT	1	
DollarToEuroRate	ExPro	currencycache	52	11-15-2016 03:02:08 GMT	3	

After the charts, details of cache keys are listed. This list provides the size of each cached value corresponding to a key and the number of times the cache was accessed. You can delete a cache manually from this list as an alternative to calling the `remove()` method in Apex. The **Delete** button lets admins manage the cache without modifying any code.

The DollarToEuroRate key uses much less space than the Rates key. That's to be expected because the DollarToEuroRate stores only one value while Rates stores an array of sObjects. Our sample app stores exchange rates for 11 currencies. If the currencies were stored individually as fields, the total size of 11 fields would be:

```
572 bytes = 52 bytes * 11
fields
```

The Rates key is a list of Exchange_Rate__c custom objects representing each a rate. Not surprisingly the list of all 11 sObjects is larger than 572 bytes (it is 1,073 bytes) because our sObject contains more fields. The additional fields include the base currency, target currency, and system date fields.

Use the information on the diagnostics page to determine whether to adjust your cache usage. For example, if a cache value has a low access count, indicating that it's rarely used, consider whether it's really needed, especially if it's large. Or if you're getting close to the cache usage limit, reconsider what to cache. You might want to remove unnecessary cache values or purchase more cache capacity.

Congratulations! You've learned how to use Platform Cache to cache your app's data for fast retrieval. You've also learned about caching best practices and how to diagnose cache usage. You're now ready to join the adventures of the chipmunk and cache precious resources!

Resources

Get Started with Event Monitoring

Learning Objectives

After completing this unit, you'll be able to:

- Name several event types supported by event monitoring.
- Define event log files.
- State at least three use cases for event monitoring.

- Describe the API-first approach to development.

What Is Event Monitoring?

Everyone knows that being a detective is one of the coolest jobs you can have. Well, hold on to your magnifying glass because your job as a Salesforce administrator is about to get a whole lot cooler. With event monitoring, you can be the investigator your organization always needed.

Event monitoring is one of many tools that Salesforce provides to help keep your data secure. It lets you see the granular details of user activity in your organization. We refer to these user activities as *events*. You can view information about individual events or track trends in events to swiftly identify abnormal behavior and safeguard your company's data.

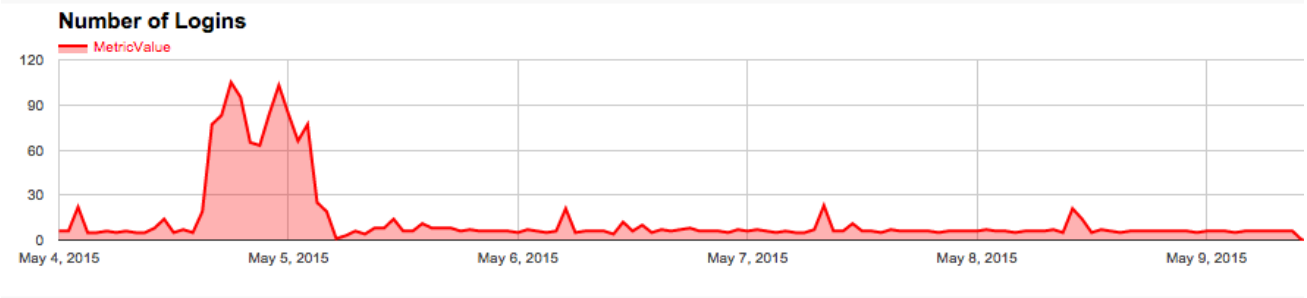
So what are some of the events that you can track? Event monitoring provides tracking for more than 30 different types of events, including:

- Logins
- Logouts
- URI (web clicks)
- UITracking (mobile clicks)
- Visualforce page loads
- API calls
- Apex executions
- Report exports

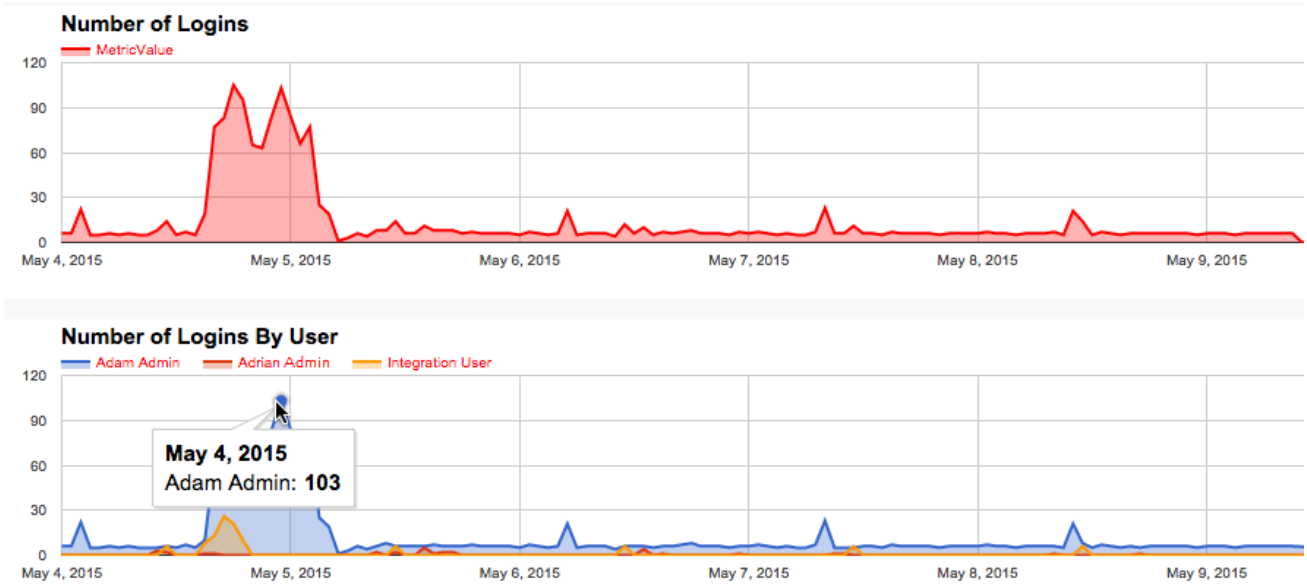
All these events are stored in *event log files*. An event log file is generated when an event occurs in your organization and is available to view and download after 24 hours. The event types you can access and how long the files remain available depends on your edition.

- Developer Edition (DE) organizations have free access to all 30+ log types with one-day data retention.
- Enterprise, Unlimited, and Performance Edition organizations have free access to the login and logout log files with one-day data retention. For an extra cost, you can access all log file types with 30-day data retention.

So how can you use event log files to become an all-knowing Salesforce super-sleuth? Let's take login activity as an example. We'll talk about accessing, downloading, and visualizing event log files later on. For now, assume that we did these steps and produced this graph of login activity.



You can see that an unusually high number of logins to the organization occurred between May 4 and May 5. But how do you figure out exactly what happened during that time period? Luckily, event monitoring provides several ways for you to dig into this data. In this case, you might want to break down the number of logins by user.



Adam Admin logged in 103 times! Something is definitely fishy. You can continue to break this data down to see things like how many distinct IP addresses a user logged in from. This information helps you pinpoint whether an outside party compromised a user's account or whether a user is up to no good.

You're probably beginning to see the power of event monitoring, but let's consider some other uses.

- **Monitor data loss**—Imagine that a sales representative leaves your company and joins a major competitor. Later, you find out that your organization is losing deal after deal to this other company. You suspect that your former employee downloaded a report containing leads and shared it with the competition. If you'd been using event monitoring, you could have caught this bad behavior before it cost your company sales.
- **Increase adoption**—Event monitoring isn't just for catching your users' bad behavior. It can also alert you to parts of your organization that aren't performing well. For example, you just rolled out a new Visualforce page in your organization that combines accounts and contacts and allows end users to add custom fields. Without any metrics, it's difficult to tell how users are interacting with this page—if at all. Event monitoring helps you figure out which parts of your organization need increased adoption efforts and even helps you identify areas that need redevelopment.
- **Optimize performance**—Sometimes it's hard to determine the cause of slow page performance in your organization. Imagine that your company has an office in San Francisco and one in London. The users in London tell you that their reports are running slowly or even timing out. You can use event monitoring to determine whether the cause is related to a network issue in London or with the way your app is configured.

These cases are just a few ways that you can use event monitoring to keep your organization secure and running smoothly. Check out all the event types to discover what else you can do.

A Quick Note About the API

If you're an administrator, working with the API can be daunting. We won't go over all the nitty-gritty details in this module, but let's take a minute to review some basics. API stands for *Application Programming Interface*. You can think of it as a bridge between an application (in our case, Salesforce) and the database. Two important terms to remember when working with the API are:

- **Objects**—Almost every object in the user interface is also an object in the API (for example, Account or Case). The API also has several objects that you can't use in the user interface.
- **Fields**—The fields you're used to seeing in the user interface are also fields in the API (for example, the Account Name field in the user interface becomes the Name field in the API).

Sometimes, the user interface doesn't provide you with every possible access point to your data. That's why the API is so important. Salesforce encourages what's called an *API first* approach to development. API first means that, before you develop an application's user experience, you want to pay attention to the underlying API. The API lets you use your data in ways that aren't possible in the user interface. Considering the API in the initial planning stages lets you develop a more robust application.

Event monitoring is an API-only feature. It isn't anywhere in the Setup area. Instead, each organization's event log files are stored in an API object called EventLogFile. Salesforce provides an API tool called Workbench that lets you access your EventLogFile objects. If all this information sounds a little confusing, don't worry. We'll go through everything step-by-step during this module.

Query Event Log Files

Learning Objectives

After completing this unit, you'll be able to:

- Ensure that you have the correct permissions to use event monitoring.
- Log in to and navigate to several tools in Workbench.
- Query an EventLogFile object using the SOQL query editor and REST Explorer.
- Compare and contrast the SOAP and REST APIs for querying event log files.
- Describe the data type used to store the underlying log data.

Query Event Log Files in Workbench

Let's consider one of the example cases from earlier. A sales representative named Rob Burgle left your company a few weeks ago and joined a major competitor. All of a sudden, you start losing deals to this other company. You suspect that Mr. Burgle downloaded a report containing confidential lead information and shared it with his new employer. Normally, you wouldn't be able to confirm your suspicions. But with event monitoring, you can gather all the evidence you need to set the story straight. Let's look at how this process works.

Before we get started, double-check that you have the correct permissions to query event log files. Event monitoring requires the "API Enabled" and "View Event Log File" permissions.

Note

Shield Event Monitoring is available for free in Developer Edition orgs. All other editions require you to purchase a license.

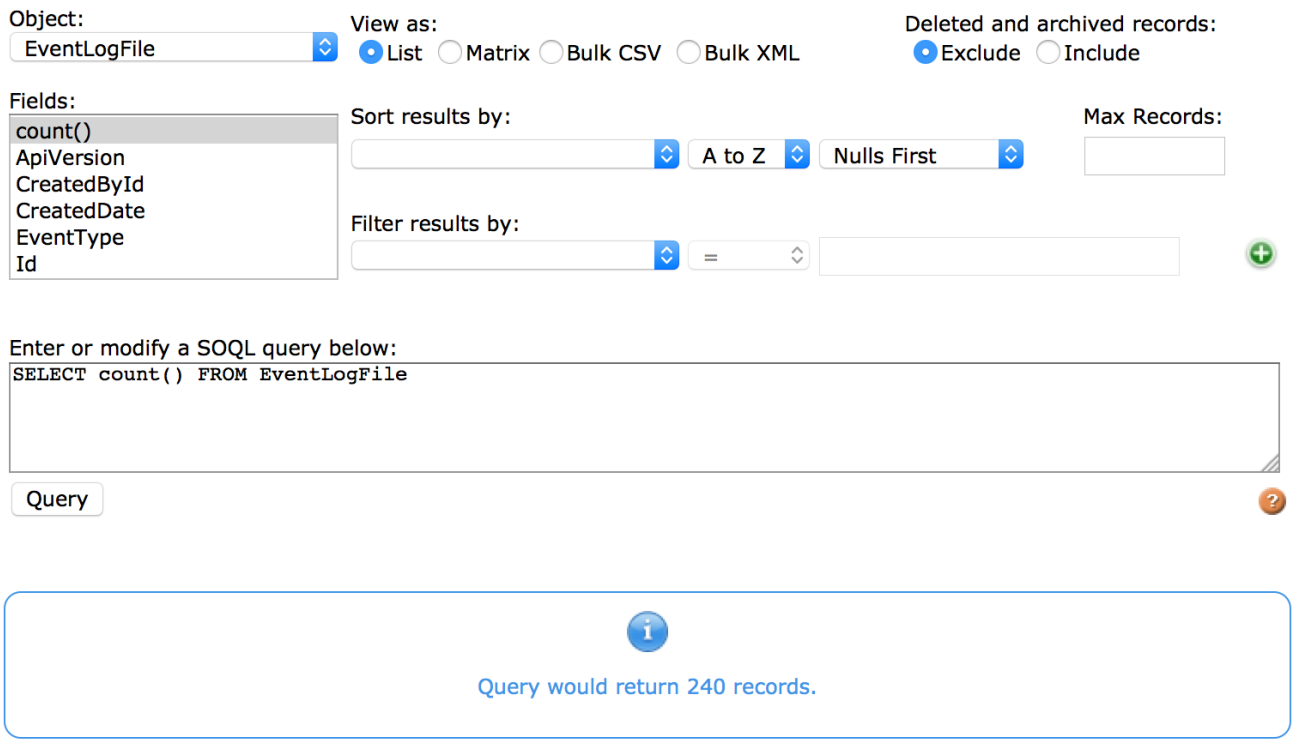
Now we're ready to go. The first step in our investigation is to log in to Workbench.

1. Log in to your Trailhead DE organization and navigate to Workbench.
2. For Environment, select **Production**. For API Version, select the highest available number. Make sure that you check **I agree to the terms of service**.
3. Click **Login with Salesforce**.

You've arrived at the Workbench home page. We don't cover all the tool's features in this module, but we go through the pieces that are useful for event monitoring. Let's start by using the SOQL query editor to make sure that you have some data to work with.

1. In the top menu, select .
2. Under Object, choose **EventLogFile**. Under Fields, select **count()**. Notice that the editor populates with some query text.
3. Click **Query**.

You should see something like this:



The `count()` function returns how many EventLogFile records exist in your organization. If the response tells you “Query would return 0 records,” it means that you don’t have any stored events. Remember that it takes 24 hours for events to surface and the log files are only stored for 24 hours in DE organizations. If you don’t get any results back, you can retry tomorrow.

So what exactly does the EventLogFile object store? To find out, we can do what’s called an object *describe*:

1. In the top menu, select .
2. Select **EventLogFile** from the dropdown menu.
3. Expand the **Attributes** menu to view the object’s properties. EventLogFile is queryable, which means that you can request information about the object from the database. It’s also retrievable, so you can find an EventLogFile record by its ID.
4. Expand the **Fields** menu. There are 15 fields here, but let’s pay particular attention to two of them: EventType and LogFile.
    - EventType: This field displays which of the more than 30 event types a record represents. If you expand , you can see the different types of events. In our case, we’re interested in records with an EventType of Report Export.
    - LogFile: This field is where the actual information you’re looking for is stored. The contents of a log file depend on the EventType. For Report Export, this field stores everything from the ID of the user that exported the report to the browser and operating system that they used to do it.

We’re getting closer to finding our culprit! Let’s keep collecting evidence by using another tool in Workbench: the REST Explorer.

View Events in the REST Explorer

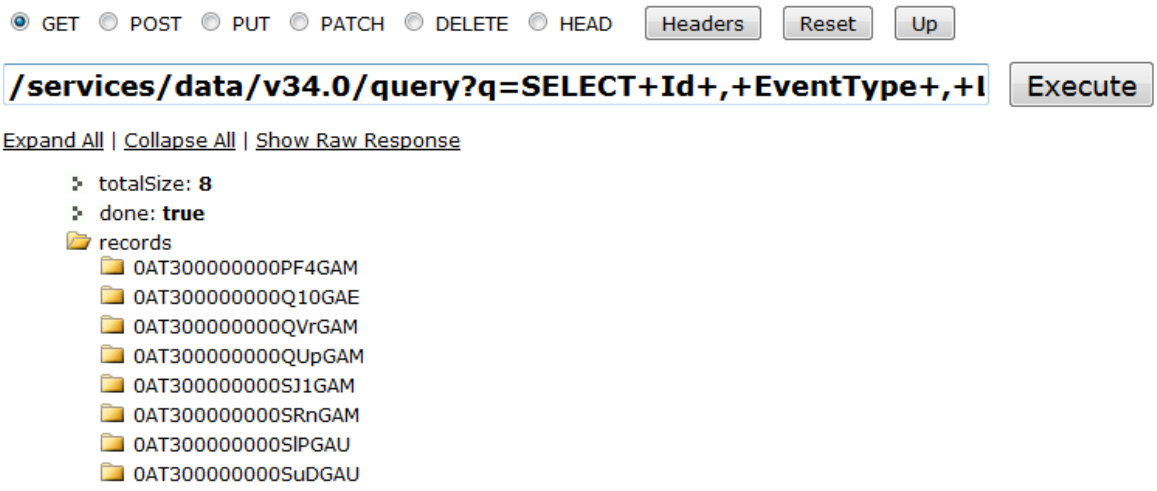
The REST Explorer gives you access to the Salesforce REST API, a web service that lets you retrieve data from your organization.

To get more information about your organization’s Report Export events in Workbench:

1. In the top menu, select .
2. Replace the existing text with `/services/data/v .0/query?q=SELECT+Id+,+EventType+,+LogDate+,+LogFileLength+,+LogFile+FROM+EventLogFile+WHERE+EventType+=+'ReportExport'`.
3. Click **Execute**.

If no reports have been exported from your organization in the past 24 hours, the totalSize field has a value of zero. Remember that it takes 24 hours for events to become available. You can export a report from your organization and try again tomorrow. Alternatively, you can replace ReportExport with a different event type in your REST query (for example, Login).

If you have some report export events, your execution returns something like this:



Expand one of the records and click the **LogFile** link. The log contents look something like this:

Yikes! How are we supposed to learn anything from this? Don't worry, we're not done yet.

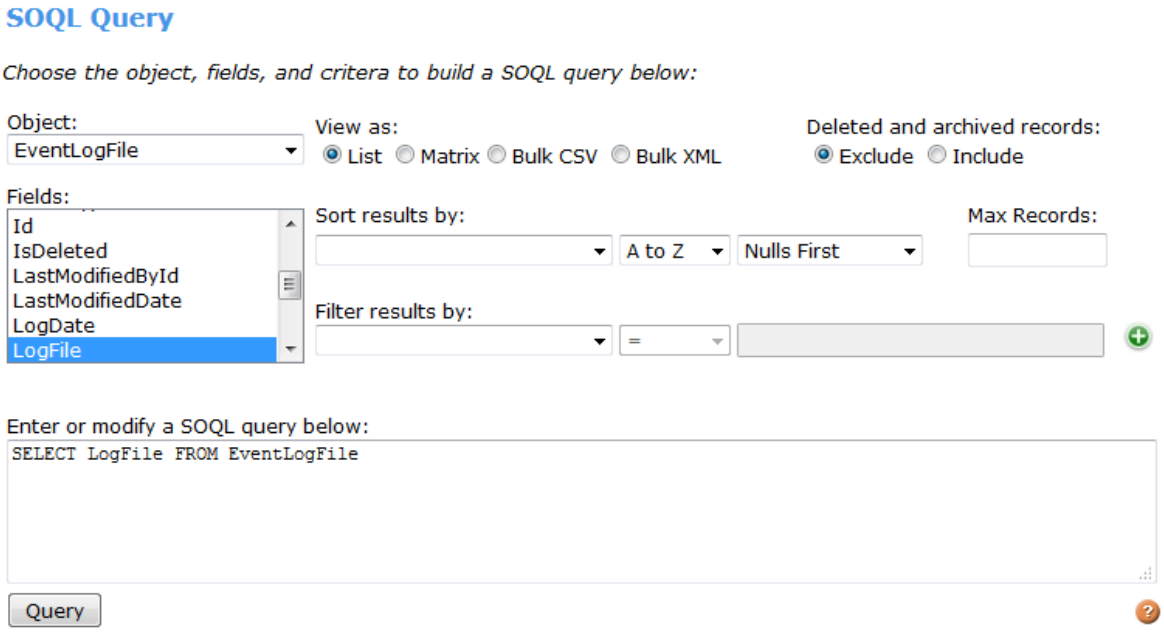
Compare Query Methods for Event Monitoring

You've used a couple of tools in Workbench. First, you used the SOQL Query Editor to determine whether you had any events stored in your organization. You also performed an object describe to learn about the EventLogFile object. Finally, you used the REST Explorer to view your EventLogFile records. All these tools retrieve information from your organization, so what's the difference between them?

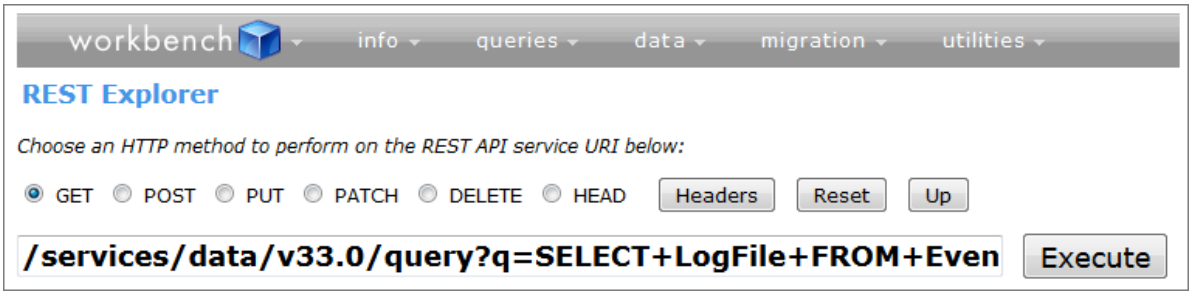
The answer isn't too surprising: The difference is in the underlying API.

The SOQL Query Editor and the object describe use what's called the SOAP API. It's a little different than the REST API that you used in the REST Explorer. One difference is that writing a query in the SOQL Query Editor is more straightforward than writing one in the REST Explorer. Let's say we wanted to retrieve one of our log files.

In SOAP, it looks like:

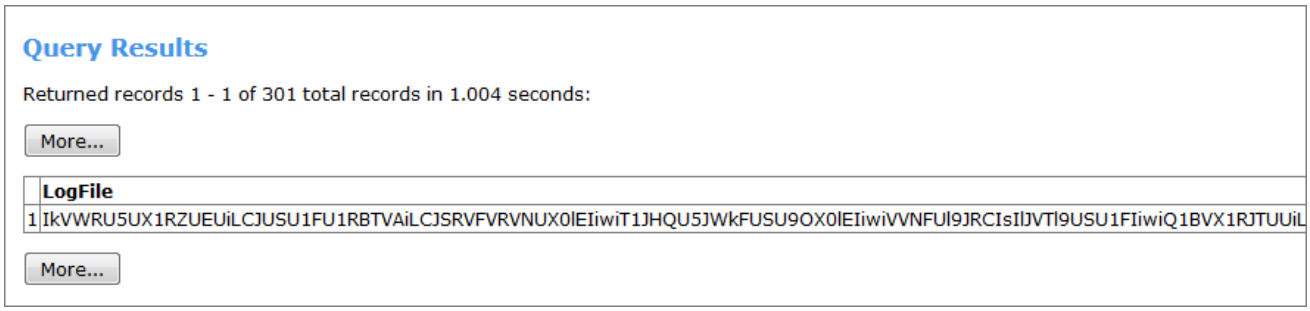


In REST, we use:

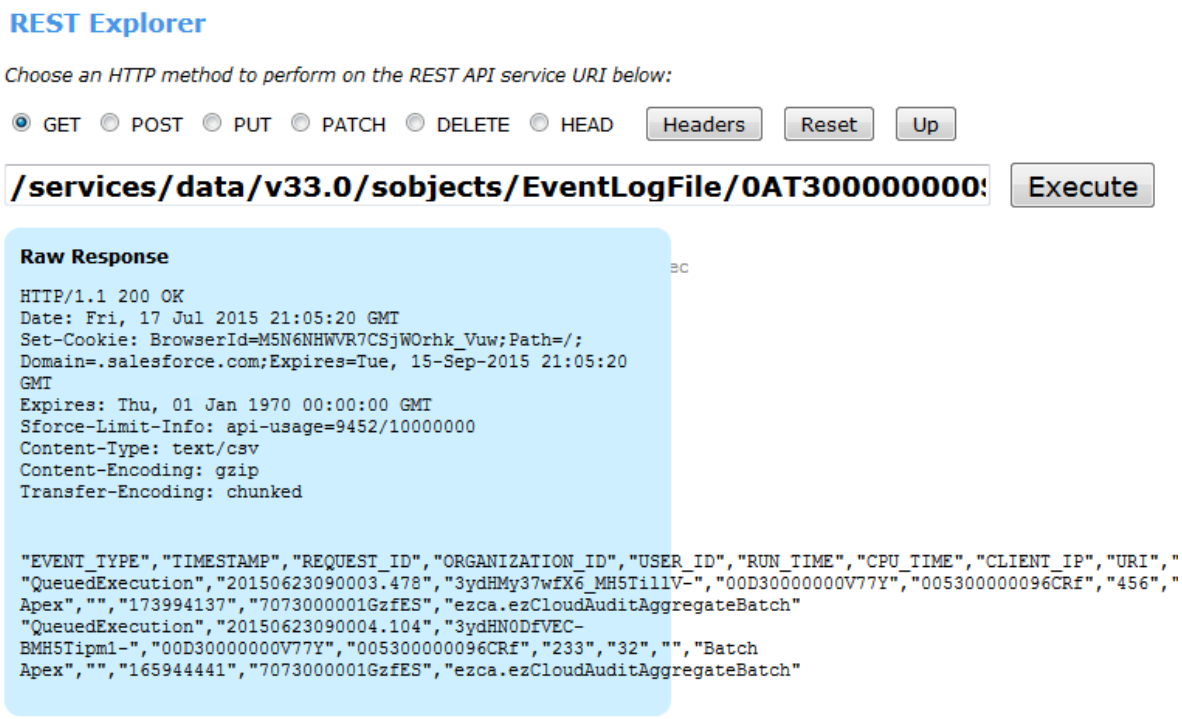


The SOQL query is easier to understand and remember. So why did we decide to use REST instead? Let's look at what happens when we execute these queries and view one of our log files.

In SOAP, the query returns something like this:



The REST query returns this:



Here, we see the other major difference between SOAP and REST when it comes to querying event log files. The returned log files are the same, but they're presented in different formats. When you retrieve your event log files using SOAP, the result is a serialized, Base64 string. If your organization plans on using tools like Informatica to work with event log files, you want to use SOAP to retrieve your data. REST, on the other hand, deserializes the log file. It's still not pretty, but as you'll see in the upcoming section, other tools can transform the REST results into an easy-to-read format.

Download and Visualize Event Log Files

Learning Objectives

After completing this unit, you'll be able to:

- Download an event log file.
- Describe the structure of event log files.
- Identify an application for downloading event log files without writing code.
- Define a role that could use a cURL or Python script for downloading data.
- Identify some options for visualizing event log file data.

Download Event Log Files

You can use Workbench to quickly check your organization's recent events and filter the events using certain criteria. But because you're accessing the data through the API, you can also use other tools that make it even easier to work with event log files. To maximize the benefits of event monitoring, you want to download your event log files from Salesforce so that you can track them over time.

You can download event log files in several ways, including:

- Direct download via the Event Log File browser application
- cURL script
- Python script

Let's look at each approach.

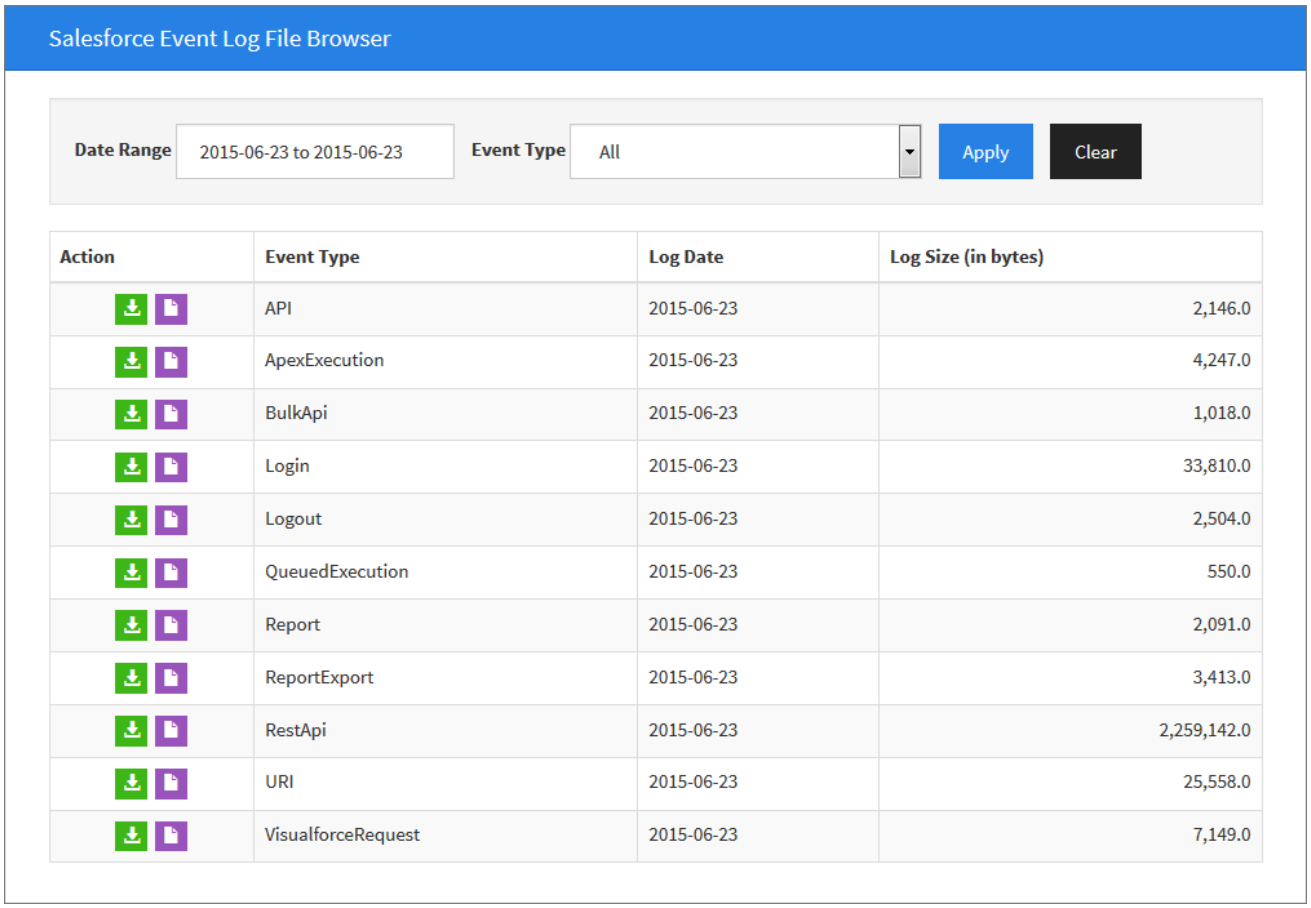
Download Logs from Your Browser

Using the event log file browser application is the most straightforward approach to downloading your organization's event monitoring data. Let's check it out.

1. Log in to your Trailhead DE organization.
2. Navigate to the event log file browser application.
3. Click **Production Login**.

4. Enter a date range for your search.
5. Enter an event type for your search.
6. Click **Apply**.

You'll see something like this:



Action	Event Type	Log Date	Log Size (in bytes)
	API	2015-06-23	2,146.0
	ApexExecution	2015-06-23	4,247.0
	BulkApi	2015-06-23	1,018.0
	Login	2015-06-23	33,810.0
	Logout	2015-06-23	2,504.0
	QueuedExecution	2015-06-23	550.0
	Report	2015-06-23	2,091.0
	ReportExport	2015-06-23	3,413.0
	RestApi	2015-06-23	2,259,142.0
	URI	2015-06-23	25,558.0
	VisualforceRequest	2015-06-23	7,149.0

Note

If your organization doesn't have any events in the specified date range or type, the page displays an error.

The list shows the same event log files that you see when you query the EventLogFile object using the REST Explorer in Workbench. You can't open the files in the browser application, but you can directly download them or use a script. Let's look at the direct download method.

Click the button to download a log to a comma-separated value (.csv) file. The ugly string of text that you saw in the REST Explorer is transformed into a format that's easily readable in a spreadsheet application, like Microsoft Excel or Google Sheets. Each file contains all the events of a particular type that occurred in your organization in the past 24 hours.

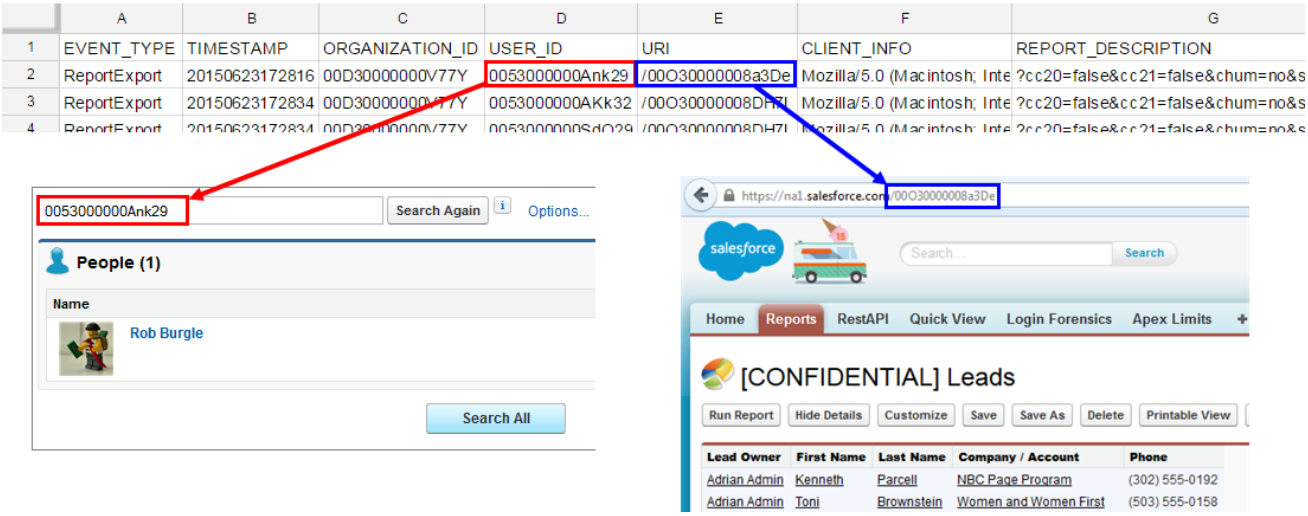
Download the ReportExport log file. Open it in a spreadsheet, and let's see what we can find.

Note

If you don't have any report export events, download another type of event log file or export a report and try this step again tomorrow. Events do not appear in the log file until 24 hours after they occur.

	A	B	C	D	E	F	G
1	EVENT_TYPE	TIMESTAMP	ORGANIZATION_ID	USER_ID	URI	CLIENT_INFO	REPORT_DESCRIPTION
2	ReportExport	20150623172816	00D30000000V77Y	0053000000Ank29	/00O30000008a3De	Mozilla/5.0 (Macintosh; Inte	?cc20=false&cc21=false&chum=no&s
3	ReportExport	20150623172834	00D30000000V77Y	0053000000AKk32	/00O30000008DH7I	Mozilla/5.0 (Macintosh; Inte	?cc20=false&cc21=false&chum=no&s
4	ReportExport	20150623172834	00D30000000V77Y	0053000000SdO29	/00O30000008DH7I	Mozilla/5.0 (Macintosh; Inte	?cc20=false&cc21=false&chum=no&s

That looks much better! Now we can finally figure out how that confidential information got leaked. Let's say that our lead report's ID is 00O30000008a3De. The URI field contains the ID of the report that was exported, and the USER_ID field contains the ID of the user who exported that report. All this information helps you pinpoint the culprit.



The user ID and the report ID are a match! You now have enough evidence to confirm that Rob Burgle exported the report. Now it's time for justice to be served!

Download Event Log Files Using cURL

We know that you're excited about cracking your first case, but this victory is only the beginning of your illustrious career as a Salesforce administrator/detective. Each event type also has a button that downloads a cURL script that you can run in your computer's command line. cURL is one of many command-line tools that you can use to download data from your organization. The script downloads a .csv file exactly like the one you downloaded in the previous step. So why use cURL instead of the direct download tool?

Although using cURL is more complicated than the first method, it provides additional flexibility in working with event log files. Rather than manually downloading log files, you can schedule when to run the script so that you always have the most recent event log files for your organization. You can also transform your data so that it's in a format that you want. If your organization has an integration specialist you can pass off these scripts to kickstart automation efforts.

Note

cURL is best-suited for Mac and Linux users. It's possible to use it on Windows, but it requires additional configuration.

Using a cURL script to download your event log files requires the following:

1. Providing your Salesforce credentials
2. Logging in using oAuth and getting an access token
3. Using a REST query to specify which logs you're looking for

Note

If you're scheduling a recurring download, this step is important. You can use something like this query to filter events by the current day.

```
https://${instance}.salesforce.com/services/data/v34.0/query?
q=Select+Id+,+EventType+,+LogDate+From+EventLogFile+Where+LogDate+=+${day}
```

4. Parsing the results of the query so that you can do things like create a date-based file structure—you can perform any transformations on your data that you want

For more information on using cURL with event log files, see this post.

Download Event Log Files Using Python

If you need a more programmatic way of downloading your organization's event log files, you can use Python scripts. One advantage of using a Python script over a cURL script is that it's easier for Windows users to work with, but it's also suitable for Mac and Linux users.

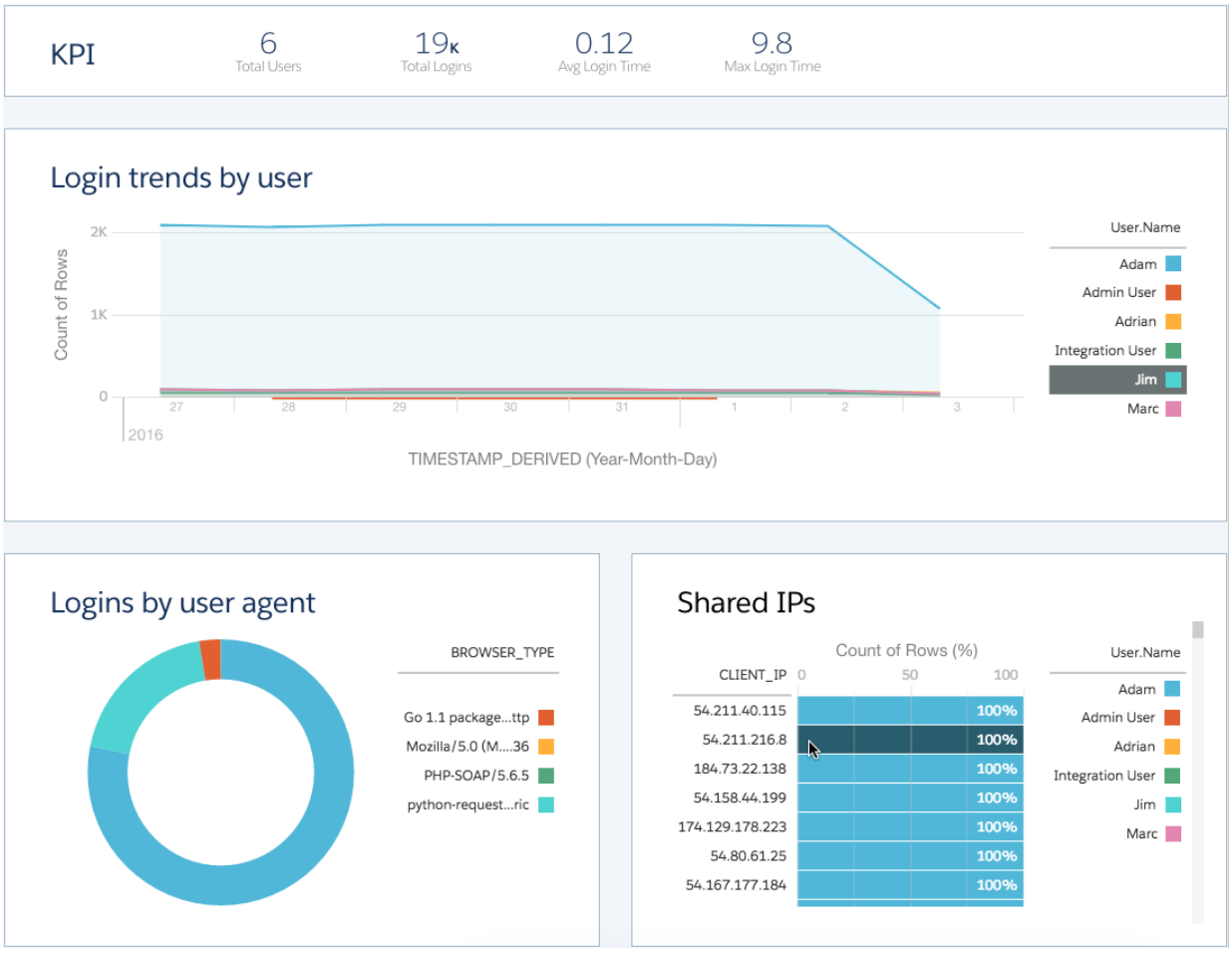
Python is easy to understand, even if you're not an experienced programmer. Some setup is required, but after that you can easily run your download script. For more information and to download the code, see this post.

Visualize Event Log File Data

Now that you've taken the time to learn about event log files and how to download them from Salesforce, it's time to talk about visualization. Searching for a specific piece of information in thousands of rows in a spreadsheet is like searching for a needle in a haystack. Most of the time, it's not useful to look for a single instance of a report export or user login. You're probably more interested in noticing behavior that's out of the ordinary. To get immediate insights into your organization's inner workings, you can regularly download your event log files and create visual representations of your data.

Event monitoring doesn't come with a visualization tool, but several existing tools are available to beautify your data. Some provide specific support for event log files, while others require more setup. We won't go into the details of each platform, but check out this list for some ideas.

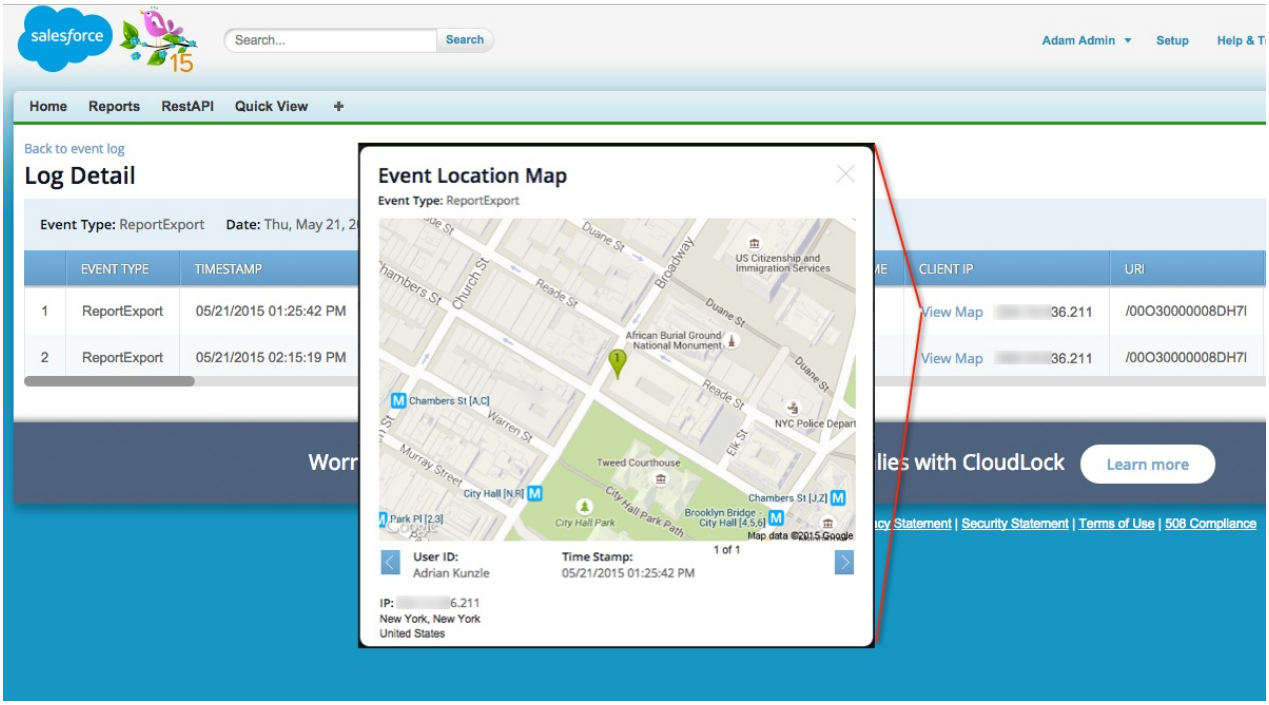
- **Event Monitoring Wave App**—The Event Monitoring Wave app for the Wave platform is a way to get insights into your event monitoring data without ever leaving the platform. Your data is automatically loaded from Salesforce to the app so you always get the most recent (and most stunning) visualization of what's going on in your org. The app provides a collection of dashboards that use pre-integrated event data, so it's a great way to get started with event monitoring.



Note

As part of Event Monitoring, you also get the Event Monitoring Wave App. Use this app to upload and access only the data provided to you as part of your subscription. Please prevent your users from using the app to upload or access any other data. Salesforce sometimes monitors such usage. The Event Monitoring Wave App is available in English only.

- **Splunk App for Salesforce**—The app lets you analyze and visualize your organization's use of Salesforce and gain insights into security, performance, and user behavior.
- **CloudLock and CloudLock Viewer**—CloudLock, a cloud security provider, offers CloudLock for Force.com and the free CloudLock Event Monitoring Viewer. These tools give security professionals and internal auditors insight into event log files.



- **ezCloudAudit**—Transforms event log files into easy-to-read tables and presents aggregated data in dynamic charts and graphs. ezCloudAudit tracks everything a user views in Salesforce, enabling you to drive user adoption, identify employee transgressions, and streamline auditing and compliance.
- **SkyHigh Networks**—Provides a network proxy solution for cloud discovery and integrates with event log files to enrich user activity tracking for security and compliance concerns.
- **FairWarning**—Protects sensitive information held in Salesforce by centrally applying security and governance policies across all your Salesforce organizations. FairWarning installation takes 30 minutes from the AppExchange.
- **New Relic Insights**—This solution for Salesforce makes it simple to understand newly available rich data about your organization's activity. Automatically import your event monitoring data into Insights to power your easy-to-build dashboards and instantly query your data in the user interface.
- **Palerra**—The LORIC platform from Palerra secures data within Salesforce using machine learning and data science to analyze event log files and recognize anomalous user behavior. It enables administrators to manage auditing and compliance while increasing operational efficiency.

You now have an idea of what event monitoring can do for your organization. You've used event log files to solve a case and seen the many possibilities for downloading and visualizing your organization's events. Now you have the tools you need to investigate, secure, and improve your organization. Good luck, detective.

Resources

Getting Started with Custom Metadata Types

Learning Objectives

After completing this unit, you'll be able to:

- Define custom metadata types.
- Identify several use cases for custom metadata types.
- Describe the structure of a custom metadata type.

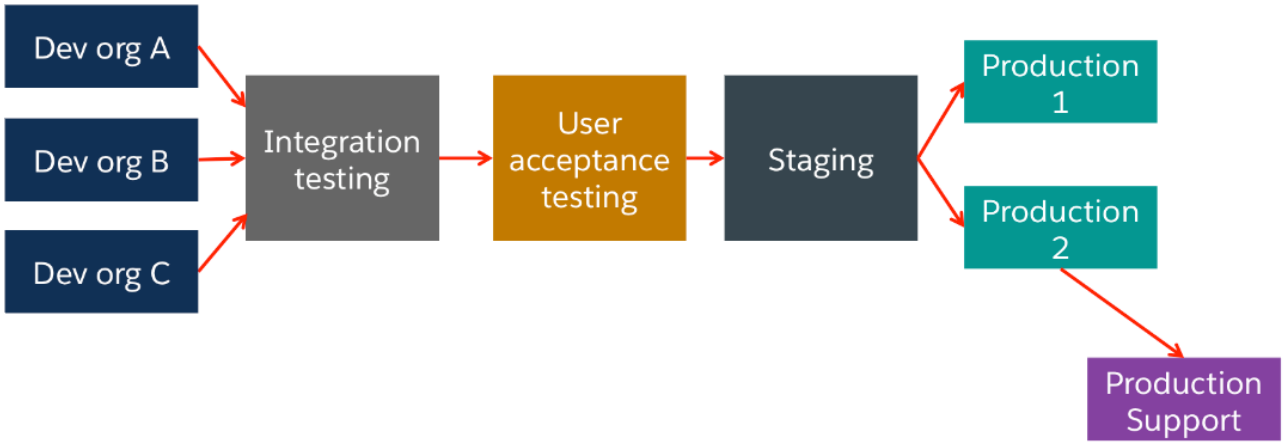
Getting Started with Custom Metadata Types

Hey, there. You look like the adventurous type. You live for the thrill, constantly seeking that ever-elusive adrenaline rush. That's how you landed this sweet job as a Salesforce developer for Relaxation Gauntlet, the world's premier adventure vacation provider.

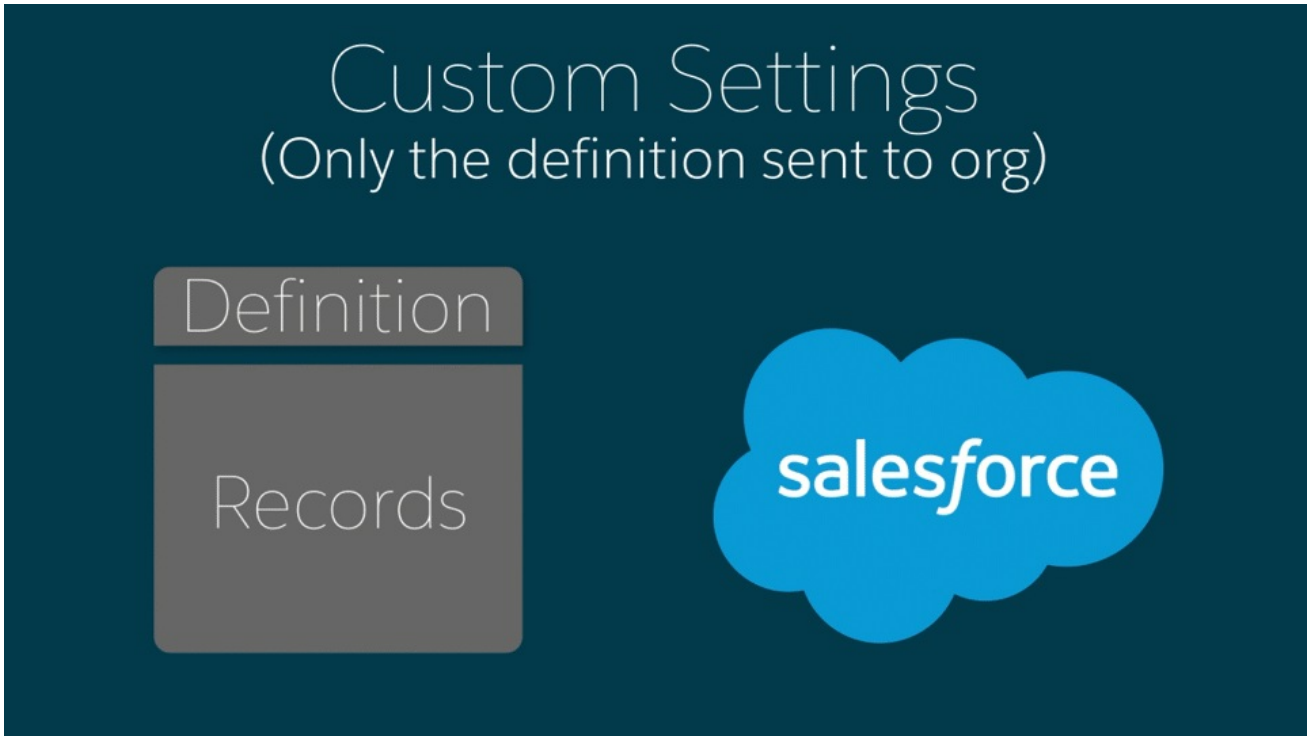
But your work there isn't all about rappelling down cliffs and chasing cheetahs through the grasslands. In the past year alone, you've dedicated hundreds of development hours to one thing: migrating app configuration records between Salesforce orgs.

What do we mean by app configuration records? Basically, these records don't store business data. Instead, they customize your app's behavior. For example, when you create a workflow rule, you're really creating a record in Salesforce that tells your app to behave a certain way. Workflow rules are a standard Salesforce configuration offering, but you can also create your own custom objects with configuration records that define behavior.

Some companies don't rely heavily on custom configuration records. They can get away with using custom settings or a limited number of custom object records. But for most Salesforce developers, using lots of custom object records for app configuration is a standard practice. If you've taken this approach before, you're probably familiar with the process.



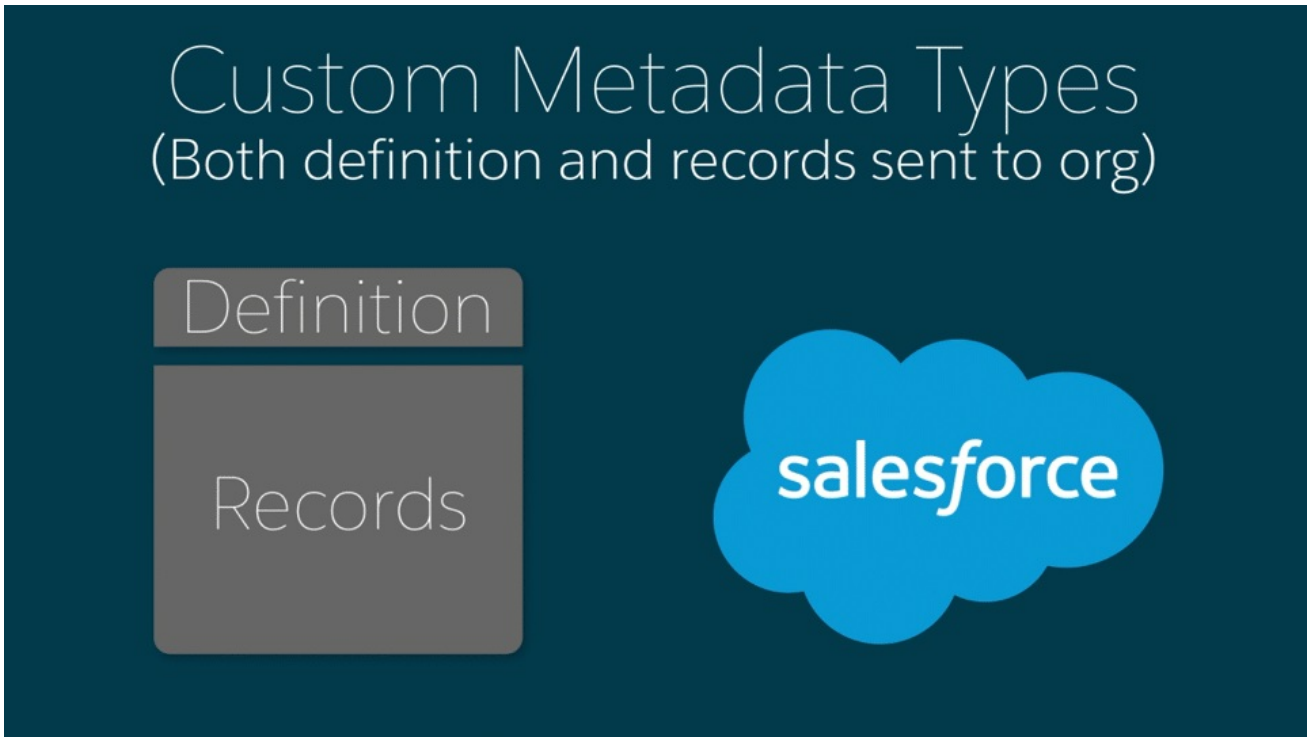
This process involves creating configuration records in one or more development orgs and then sending them through integration and user acceptance testing. Finally, you deploy them to at least one production org. Throughout all these steps, you're stuck doing a tedious manual transfer of your records. And for someone who thirsts for excitement, this situation isn't ideal.



Well buckle up and get ready for something crazy: *custom metadata types*.

That's right. You didn't think it was possible, but your life at Relaxation Gauntlet is about to get even wilder. With custom metadata types, you can customize, deploy, package, and upgrade application metadata that you design yourself.

Put more simply, custom metadata types let you use records to configure your app without worrying about migrating those records to other orgs. You can deploy the records of custom metadata types from a sandbox with change sets or packaged in managed packages instead of transferring them manually. More time to try out that new underwater jet pack!



We know you're starting to feel that adrenaline pump. After all these years of using the same old Salesforce metadata types, you finally get to spice things up with some of your own.

Using Custom Metadata Types

Custom metadata types are very, well, meta. So before we dive in any deeper, let's take a look at some concrete examples of what you can do with custom metadata types.

- **Mappings**—You can use custom metadata types to create associations between different objects. For example, you can create a custom metadata type that assigns cities, states, or provinces to particular regions in a country.
- **Business rules**—Salesforce has lots of ways to define business rules. One way is to combine configuration records with custom functionality. For example, you can use custom metadata types along with some Apex code to route payments to the correct endpoint.
- **Master data**—Say that your org uses a standard accounting app. You can create a custom metadata type that defines custom charges, like duties and VAT rates. If you include this type as part of an extension package, subscriber orgs can reference this master data.

Sweet! So we know some things that custom metadata types are used for, but how do they actually work?

More About Custom Metadata Types

When you create a custom metadata type, you also create custom fields on that type. Let's consider a simple master data scenario: You want records that store the name of each country where your company sells vacation packages and their associated country codes. First, you create a custom metadata type called Country Code Data. Next, you add fields to that type. Custom metadata types support most standard field types, including:

- Metadata Relationship
- Checkbox
- Date and Date/Time
- Email and Phone
- Number
- Percent
- Picklist

- Text and Text Area
- URL

For our scenario, we want to create two custom text fields on our type: Country Name and Country Code.

After you've created the type with all its fields, you want to create custom metadata records that are defined by that type. These records store the actual configuration data. For our country code example, you'd create a record for each country that's relevant to your org.

Table 1. Country Code Data (Custom Metadata Type)

Master Label	Country Name (Custom Field)	Country Code (Custom Field)
Austria	Austria	AT
Belgium	Belgium	BE
Denmark	Denmark	DK
...	...	...

You can also define reusable functionality that determines your application's behavior based on the configuration defined in your custom metadata type. For example, you could write an Apex method that automatically replaces the country name with the country code in a Contact's address field.

You can then package up and install all these components, including your records, in other orgs. If you need to include more country codes later on, you can add the new records to your package.

Look at you, just hanging around reading instead of jumping into the action where you belong. All that's about to change. Let's journey onward and start creating some types of our own!

Creating and Managing Custom Metadata Types

Learning Objectives

After completing this unit, you'll be able to:

- Declaratively define custom metadata types.
- Add and edit metadata records.
- Access custom metadata types with code.
- Test your custom metadata types.

Creating Custom Metadata Types

Relaxation Gauntlet offers a wide variety of adventure vacations, all with varying degrees of danger. When sales reps open a new account, they give that account a tier (Bronze, Silver, or Gold) based on the customer's past experiences with adventure. After all, you don't want to send a first-time thrill-seeker out to dive into shark-infested waters or wrestle rodents of unusual sizes.

As the Salesforce developer, you've been charged with creating an app configuration that defines which vacations are available for each account tier. You also need to deploy this app to Relaxation Gauntlet's production org with all its configurations. Sounds like a job for custom metadata types.

1. Search Setup for `Custom Metadata Types`.
2. Click **New Custom Metadata Type**.
3. Call your new type Threat Tier.

After you save, you land on the Threat Tier detail page where you can do things like add custom fields and edit page layouts. We want our custom metadata type to associate each vacation with an account tier, so we need two custom fields.

Adding a custom field to a custom metadata type is just like adding a custom field to a standard or custom object. Simply click **New**, select **Text**, and then add a label. Create an Account Tier field with a 10-character limit, and add a Vacation field with a 40-character limit. You can leave the rest of the default options.

Believe it or not, you finished creating your custom metadata type. Let's keep up the pace and start learning about custom metadata records.

Creating Custom Metadata Records

By itself, a custom metadata type isn't that useful. The whole point of custom metadata types is that you can create records that are defined by a type. So enough waiting around. Let's do it!

1. On the Threat Tier detail page, click **Manage Threat Tiers**.
2. Click **New**.
3. Call your record Bronze Option A.
4. For Account Tier, enter `Bronze` . For Vacation, enter `Deep Sea Beta Fishing`.

Repeat this process until you have all these values.

Table 1. Threat Tier

Master Label	Account Tier (Custom Field)	Vacation (Custom Field)
Bronze Option A	Bronze	Deep Sea Beta Fishing
Bronze Option B	Bronze	Base-two Jumping
Silver Option A	Silver	Ziplining Over the Great Seg Fault
Silver Option B	Silver	Scaling Mt. PaaS
Gold Option A	Gold	Bug Bounty Hunting
Gold Option B	Gold	Hiking the App-alachian Mountains

You can create these individual records and build some custom functionality using SOQL and Apex to get your app working the way you need it to.

Let's be honest: If you're like most adrenaline junkies, all this one-by-one record creation isn't sounding very extreme. Lucky for you, we built a tool that lets you kick things up a notch. If you're interested in creating a bunch of records at once, the Custom Metadata Loader is for you. You can upload up to 200 custom metadata records to your org from a CSV file or update up to 200 existing records. We won't do this right now, but check it out if you're working with a large number of configuration records.

Developing with Custom Metadata Types

In the next unit, we'll talk about the specifics of who and what can modify your custom metadata types and records. But before we do that, let's talk about something simple: using SOQL and Apex to access your types and records.

Throughout your many wondrous adventures into Salesforce development, you've probably defined a lot of custom object records in Apex. Lucky for you, a similar convention applies to custom metadata types. You can declare an Apex variable of a custom metadata type like so.

```
Threat_Tier__mdt
Bronze_Option_A;
```

Think you can make the jump to getting the values of a custom field from all the custom metadata records of a certain type? Notice that even though the suffix for the custom metadata type is `__mdt`, the suffix for the custom field stays as `__c`.

```
Threat_Tier__mdt [] vacationNames = [SELECT Vacation__c FROM
Threat_Tier__mdt];
```

See, you're already a master. You can use custom metadata types in Apex the same way you use any standard Salesforce configuration object. But you have the added bonus that you can quickly transport records from development to subscriber orgs!

Testing Custom Metadata Types

Typically, Apex tests don't depend on the existence of particular records. Because your org's functionality doesn't rely on specific accounts or opportunities, you don't need to write tests for them. But when you're dealing with records that affect the way an org behaves, you want to ensure that everything's working as expected.

If you've tested apps that use custom objects to hold app configuration records in the past, you might be familiar with the `@IsTest(SeeAllData=true)` annotation in Apex. This dangerous line of test code lets your Apex tests see all the records in your org, including data records like accounts and contacts. You might be wondering whether you need to take advantage of this bad practice to test your custom metadata types.

Luckily, that's not the case. Custom metadata types are set up the same way as things like workflow and validation rules. Your Apex test classes can see them and access their fields and records. Ultimately, your test cases look more or less like any other Apex test. For more advanced cases, check out the blog post in the resources.

Resources

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Protecting Custom Metadata Types and Records

Learning Objectives

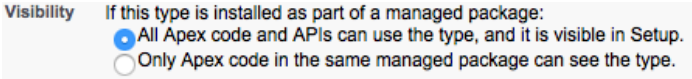
After completing this unit, you'll be able to:

- Protect custom metadata types.
- Control the editability of a field.
- Protect custom metadata records.

Protecting Custom Metadata Types

All good adventurers know that safety is a top priority. The same goes for your custom metadata types and records—you need to control access to keep your apps functioning as intended.

When we created our Threat Tier type, you might have noticed that we skipped over a section of the interface.



When you create a type, you can choose whether the type is public or protected. All Apex code in an org can use public types, while the use of protected types is restricted to code inside the package itself.

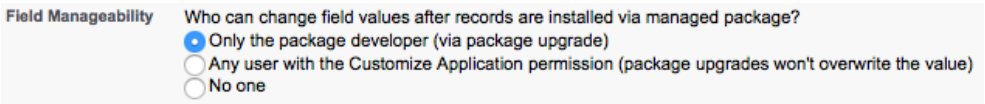
That's pretty simple. It gets a little rockier when we start to talk about controlling edit access to custom metadata fields and records.

Protecting Custom Metadata Records

Instead of protecting the entire metadata type, you can leave the type public and protect individual records. A protected record is only accessible to code in the same namespace as the record or its associated custom metadata type. But because the type is public, a subscriber org or extension package can add its own records to that type.

Field Manageability

When it comes to protecting fields on custom metadata types, you have three options.



- The package developer can edit the field after release via package upgrades. Subscriber orgs can't change the field.
- The subscriber org can edit the field after installing the package. Package upgrades don't override the subscriber's changes.
- The package developer nor the subscriber can edit the field after the package is released.

All these options might not seem that complicated, but let's take some time to dig in to how everything works together.

Putting It All Together

Let's say we're putting our Threat Tier type in a managed package.

For legal reasons, we don't want package subscribers to change the Account Tier field. As the package developer, you still want to be able to change it in later package releases in case the legal requirements change. So, you want the Account Tier field to be upgradeable.

On the other hand, the values on the Vacation field vary depending on the org. Sometimes orgs change the names of the vacations depending on their location. To account for this need, make the Vacation field subscriber editable.

Take a look at this table to see who can edit the various fields on both public and protected records in a managed package.

	Developer Name	Label	Account Tier (upgradeable)	Vacation (subscriber editable)
Public record	—	Developer	Developer	Subscriber
Protected record	—	Developer	Developer	—

In a managed package, the package developer can always edit upgradeable fields. Subscribers can edit subscriber editable fields on public records but not on protected ones. In fact, subscriber editable fields on protected records are essentially locked after release.

If we look at the same custom metadata type and its associated records in an unmanaged package, it's easier to keep track of who can edit what.

	Developer Name	Label	Account Tier (upgradeable)	Vacation (subscriber editable)
Public record	Subscriber	Subscriber	Subscriber	Subscriber
Protected record	Subscriber	Subscriber	Subscriber	Subscriber

An unmanaged package gives the subscriber free rein over editing records and fields on custom metadata types.

The moral of this adventure is that field manageability can get complicated very quickly. It's important to plan how you're going to manage your custom metadata types, fields, and records before you package your app and release it to the wild.

Speaking of packaging, we'll start that adventure next!

Packaging Custom Metadata Types

Learning Objectives

After completing this unit, you'll be able to:

- Package your custom metadata types.
- Package your types' associated records.
- Upload your package.
- Install your package in a subscriber org.
- Upgrade your package.

Packaging Custom Metadata Types

When you're ready to package up your custom metadata types and records, you can choose your own adventure. Managed packages, unmanaged packages, and managed package extensions are all compatible with custom metadata types. And change sets let you deploy your types and records from a sandbox.

You're itching to get your Threat Tier type out to your other orgs, so let's do it. From Setup, search for `Packages` and then click a package name (or create a package if you haven't yet). You can use either a managed or an unmanaged package.

Note

You can have only one managed package per Developer Edition org. To create a managed package, your org must be namespaced. If you aren't familiar with namespaces or don't want to add a namespace, use an unmanaged package.

Next, add your custom metadata type to the package.

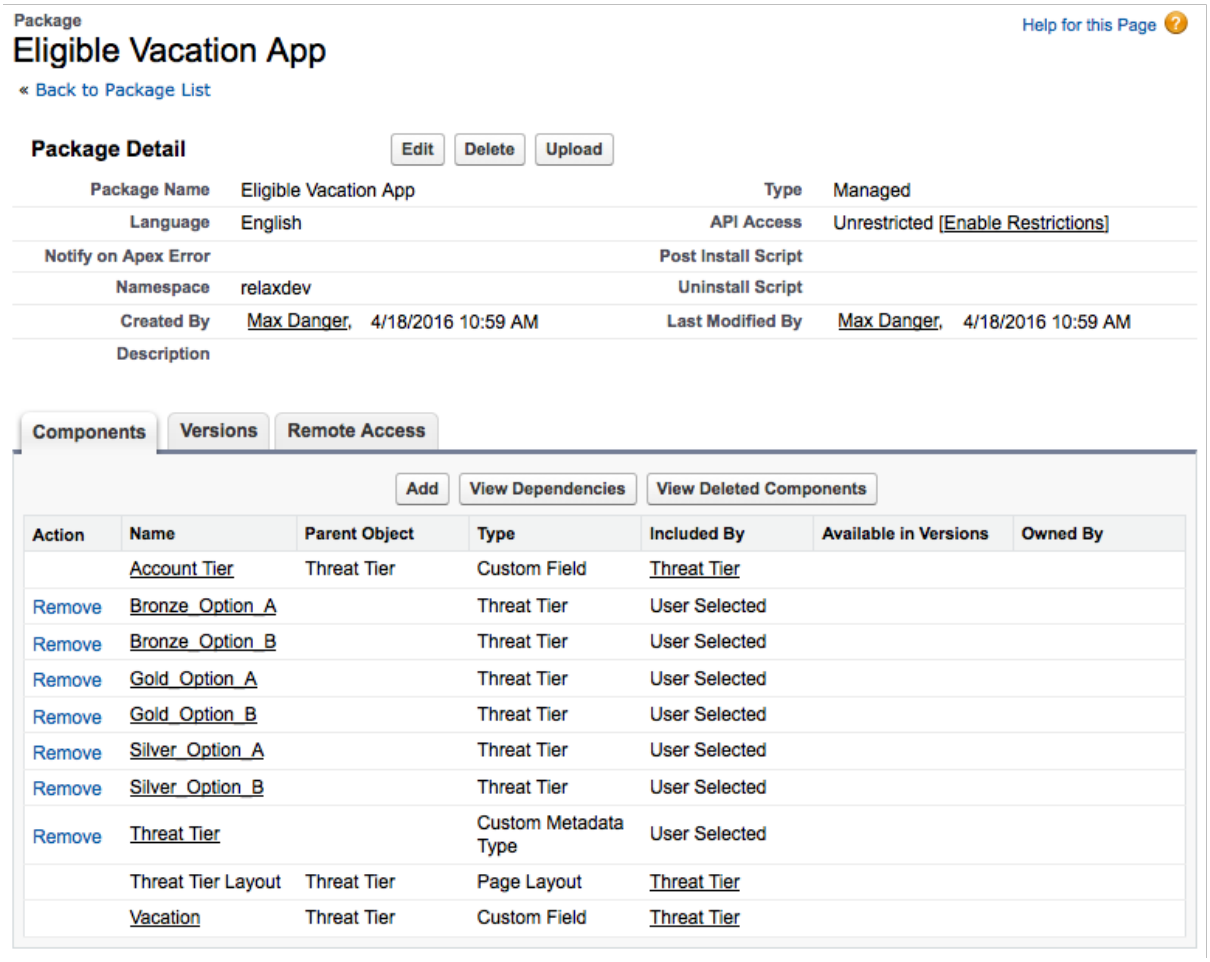
1. Click **Add**.
2. For the component type, select **Custom Metadata Type**.
3. Select **Threat Tier**.
4. Click **Add to Package**.

Easy! Notice that your custom metadata type and its fields were added to the package, but your records are missing. Let's take care of that next.

Packaging Custom Metadata Records

The process for adding custom metadata records to a package is almost the same as it is for packaging types. The only difference is that instead of selecting a standard option from the list of component types, you select your specific type. Let's give it a try, starting from the package detail page.

1. Click **Add**.
2. For the component type, select **Threat Tier**.
3. Select all your custom metadata records.
4. Click **Add to Package**.



You don't have to add a custom metadata type's records to the package. However, if you add records of that type to a package, your type is added to that package automatically.

Uploading Packages

After you've added all your custom metadata types, records, and any other components your app needs, you're ready to rock and roll. Uploading a package that contains custom metadata types and records isn't any different from uploading other types of packages.

Click **Upload** on the package detail page to kick off the process. Set the version name to `Spring 2016`, and leave the Version Number as is.

Typically if you're using a managed package, you want your first release to be beta. But we're all about diving in, so let's go straight for the big time. Select **Managed - Released** for the release type. If you're using an unmanaged package, don't worry about this part.

For more complex packages, you'll probably have to spend some time fiddling with the other options on this page. But for now, just click **Upload** and wait for the process to complete.

Installing Packages

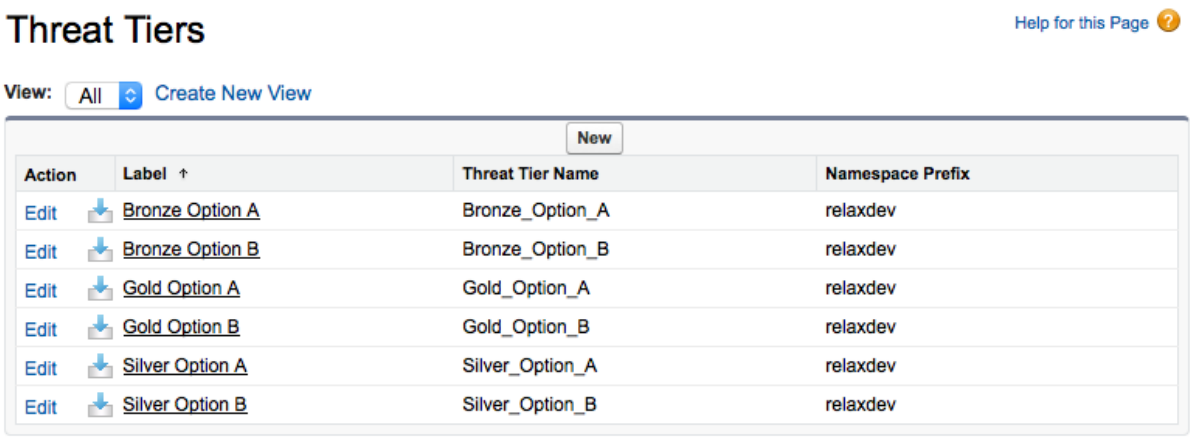
When your upload is finished, you'll receive an email with an installation link for your package. If you have another Developer Edition org, you can see for yourself the results of your fearless adventure into the world of meta. If not, you just have to take our word for it.

Make sure that you're logged out of the org where you developed the package. Follow the link in the email, and log in to your alternate account. When prompted, install the app for admins only. When the installation completes, your app appears on the Installed Packages page.

But how do we know whether we successfully completed our journey?

In the Quick Find box, search for `Custom Metadata Types`. If all went as planned, you see your Threat Tier custom metadata type. That's all well and good, but what you're really interested in is whether your *records* migrated over.

Click **Manage Records** to see the magic for yourself.



Looks like the gang's all here! You've successfully moved your app configuration records from one org to another in just a few easy steps.

Upgrading Packages with Custom Metadata Types

Of course, our adventure isn't over yet. App development is rarely a one-and-done process. Down the line, you'll want to make improvements and fix your app.

Upgrading managed packages is where our earlier discussion about field manageability is especially important.

For example, we made our Account Tier field upgradeable. So as the package developer, you can change the Account Tier field and the values of this field on records in future iterations of the app. On the other hand, your subscriber orgs can't make changes. They can create Threat Tier records, but not edit existing values.

Vacation is a subscriber editable field, so the subscriber org can change the field based on its own needs. Subscriber editable also means that you can't overwrite your subscriber's values through a package upgrade.

If you created a managed package, you can deploy an upgrade to see how this works. In your development org, make some changes to both the Account Tier and Vacation fields. Using the same process as before, upload your package and reinstall it in your subscriber org using the email link. Check the fields and records on the upgraded package. You can see the changes you made to the Account Tier field, but the Vacation field looks the same as it did before the upgrade.

You've made it to the last leg of the journey. Only one question remains: What do you do with all your existing app configurations now that you've found a better way?

Resources

Converting Custom Settings

Learning Objectives

After completing this unit, you'll be able to:

- Convert list custom settings to custom metadata types.
- Refactor your org's Apex code to support custom metadata types.

Making the Switch

One person, three days.

Yep, you heard it right. In less than a week, you and you alone can learn all about custom metadata types. You're already halfway there! You can convert up to 10 custom settings objects and refactor up to 200,000 lines of Apex code.

Note

Converting list custom settings into custom metadata types is easy. We don't recommend converting hierarchy custom settings into custom metadata types.

Follow these three steps to convert your configurations.

Step 1: Convert Custom Objects to Custom Metadata Types

The first part of this step is to retrieve your app metadata, including the custom objects you're using for configuration. Create corresponding custom metadata types for each object. You can do this step by hand or by transforming the objects' XML using something like XSLT.

Regardless of the method you use to convert your objects, make sure that you use the same field names on your custom metadata types. This way you don't have to change them in your Apex code later on.

When you're finished creating your types, you can package all your metadata and deploy it to your org.

Step 2: Replace __c with __mdt

By now you're comfortable with the idea that custom metadata types use the `__mdt` suffix instead of the classic `__c`.

For each custom metadata type that you created in the previous step, change the object reference in all your Apex classes and triggers. The easiest approach to this task is using the Find/Replace tool in Eclipse.

As a reminder, the `__mdt` suffix is only for the *type*, not for its fields. You can leave the custom fields with the `__c` suffix.

Step 3: Replace Apex with SOQL

After your metadata is deployed and your references are renamed, it's time to get in the thick of your Apex refactoring.

The first thing you need to do is replace your Apex methods, like `seeAll()`, with SOQL queries, such as `SELECT CustomField__c FROM MyMetadata__mdt`.

Now you can delete your deprecated custom objects and code that supports those objects. Make sure that you're getting rid of things like test code that creates mock records and post install scripts that install records.

With that, you're ready to deploy your new code to production to finish up the conversion! It's really that easy.

The Journey's End

Congratulations, adventurer. You've learned what custom metadata types are, how to create them, and how to package them up and migrate them to other orgs. You're ready to ditch your old app configuration methods and make more time in your life for things you *actually* want to do.

We're only scratching the surface of the power of custom metadata types. And with your fearlessness and sense of adventure, you'll become an expert in no time. Happy adventuring, Trailblazer!

Get Started with Visualforce in Salesforce1

Learning Objectives

After completing this unit, you'll be able to:

- Describe three places where Visualforce pages can be used in Salesforce1.
- Describe when it makes sense to build Salesforce1 apps with Visualforce.
- List at least three declarative tools you can use instead of creating Salesforce1 customizations with code.
- List at least three reasons to develop mobile apps in Salesforce1 instead of using the Mobile SDK.

Get Started with Visualforce in Salesforce1

You can use Visualforce pages to extend Salesforce1 and provide mobile users with the functionality and the user experience they need.

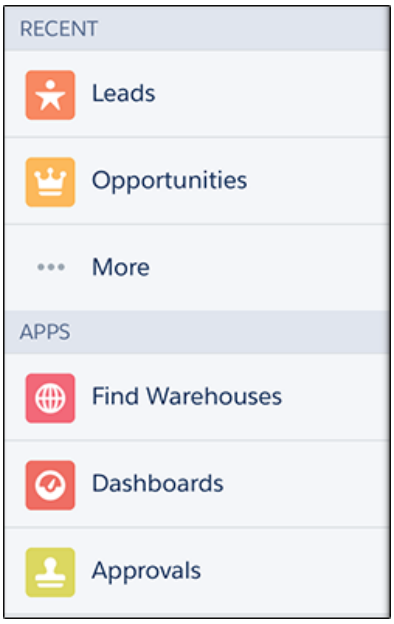
Salesforce1 provides a lot of functionality built right into the mobile application. Often, you can extend Salesforce1 for your organization's needs without writing a single line of code. But when you want to build more customized applications, you can do it with Visualforce.

Where Visualforce Pages Can Appear in Salesforce1

When you create a Visualforce page, you can make it available from a number of places in the Salesforce1 user interface.

- Navigation menu—available when you tap ☰ from the Salesforce1 mobile app

- Action bar and action menu—available from the bottom of any page in the Salesforce1 app that supports actions
- Record related information page (as a mobile card)—available when you navigate to a record



When to Use Declarative vs. Programmatic Tools to Create Salesforce1 Apps

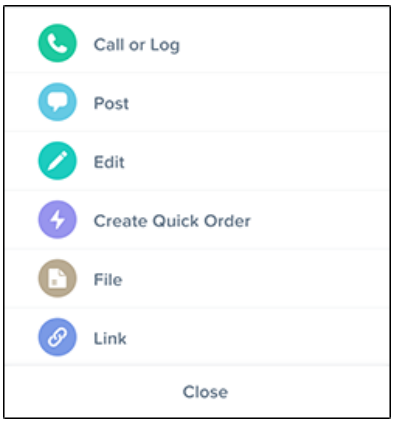
When developing functionality for your Salesforce mobile users, you can use declarative features (clicks) or programmatic tools (code). Here are some points to consider before you start coding.

Salesforce1 features built with declarative tools:

- Are usually faster and cheaper to build.
- Generally require less maintenance.
- Receive automatic upgrades when the tools are improved.
- Aren't subject to governor limits.

Common declarative tools you can use in your Salesforce1 apps include:

- Quick Actions
- Page layout customization
- Formula fields and roll-up summary fields
- Validation rules
- Workflows and approval processes
- Custom fields and objects



Before you commit to developing a new feature using programmatic tools such as Visualforce, consider whether you can implement your feature with declarative tools instead.
Of course, some apps can't be built with declarative tools. Programmatic tools are often required for features that:

- Support specialized or complex business processes.
- Provide highly customized user interfaces or customized click-through paths.
- Connect to or integrate with third-party systems.

There's no hard-and-fast rule for when to choose clicks over code. When choosing, consider the problems you're trying to solve and resources available to solve them.

When to Use the Salesforce App Cloud vs. Creating Custom Apps

When it comes to developing functionality for your Salesforce mobile users, you have options. Here are some differences between extending the Salesforce1 app and creating custom apps using the Mobile SDK or other tools.

Salesforce App Cloud

- Has a defined user interface.
- Has full access to Salesforce data.
- Can be used to create an integrated experience in the Salesforce1 app.
- Gives you a way to include your own apps/functionality with quick actions.
- Lets you customize Salesforce1 with point-and-click or programmatic customizations.
- Lets you add functionality programmatically through Visualforce pages, Force.com Canvas apps, and the Lightning Component Framework.
- Has defined navigation points. Salesforce1 customizations or apps adhere to the Salesforce1 navigation. So, for example, a Visualforce page can be called from the navigation menu or from a custom action in the action bar.
- Enables you to leverage existing Salesforce development experience, both point-and-click and programmatic.
- Is included in all Salesforce editions and supported by Salesforce.

Custom Apps

Custom apps can be either free-standing apps you create with Salesforce Mobile SDK or browser apps using plain HTML5 and jQuery Mobile / Ajax. With custom apps, you can:

- Define a custom user experience.
- Access Salesforce data using REST APIs in native and hybrid local apps, or with Visualforce in hybrid apps using JavaScript Remoting. In HTML5

apps, do the same using jQuery Mobile and Ajax.

- Brand your user interface, including having a custom app icon, for increased exposure in apps that are customer-facing.
- Create standalone mobile apps, either with native APIs using Java for Android or Objective-C for iOS, or through a hybrid container using JavaScript and HTML5 (Mobile SDK only).
- Distribute apps through mobile industry channels, such as the Apple App Store or Google Play (Mobile SDK only).
- Configure and control complex offline behavior (Mobile SDK only).
- Use push notifications (available for Mobile SDK native apps only).
- Design a custom security container using your own OAuth module (Mobile SDK only).

Other important Mobile SDK considerations:

- Open-source SDK, downloadable for free through npm installers as well as from GitHub. No licensing required.
- Requires you to develop and compile your apps in an external development environment (Xcode for iOS, Eclipse or similar for Android).

Salesforce App Cloud Development Process

The Salesforce1 app is designed to run on a mobile device, like a phone or tablet, but that doesn't mean you have to develop on one. Especially when you're working with Visualforce pages that will apper in both the Salesforce1 app and the full Salesforce site, it's convenient to develop on your desktop. By accessing the Salesforce1 app in a device emulator, you can develop iteratively on your desktop, without having to pick up your device to test each change.

Important

Running the Salesforce1 app in an emulator isn't supported for normal use, and it's not a substitute for full testing of your custom apps and pages on your organization's supported mobile devices. During development you should regularly test your app on *every* device and platform on which you intend to deploy.

To view your Salesforce1 custom pages and apps on a device emulator, download and install the appropriate SDK for your supported devices.

- Apple iOS Simulator for iPhone and iPad

  developer.apple.com/library/ios/documentation/IDEs/ Conceptual/iOS_Simulator_Guide/

- Android Virtual Device Emulator for Phone and Tablet

  developer.android.com/tools/devices/emulator.html

You can't use the standard https:///apex/*yourPageName* URL to test Visualforce in Salesforce1. You can only do that by accessing the page within Salesforce1.

Add Visualforce Pages to the Navigation Menu

Learning Objectives

After completing this unit, you'll be able to:

- Add a Visualforce page to the navigation menu.
- Add CSS styling to a page to make it mobile-friendly.
- Reference the `sforce.one` navigation library on a Salesforce1 page.
- Describe at least four different navigation actions that can be performed using the `sforce.one` navigation library.

Implement Global Actions with Visualforce Pages

Learning Objectives

After completing this unit, you'll be able to:

- Describe two differences between global actions and object-specific actions.
- Implement a Salesforce1 global action using a Visualforce page.
- Describe the Publisher SDK and list two ways to interact with publisher dialogs.

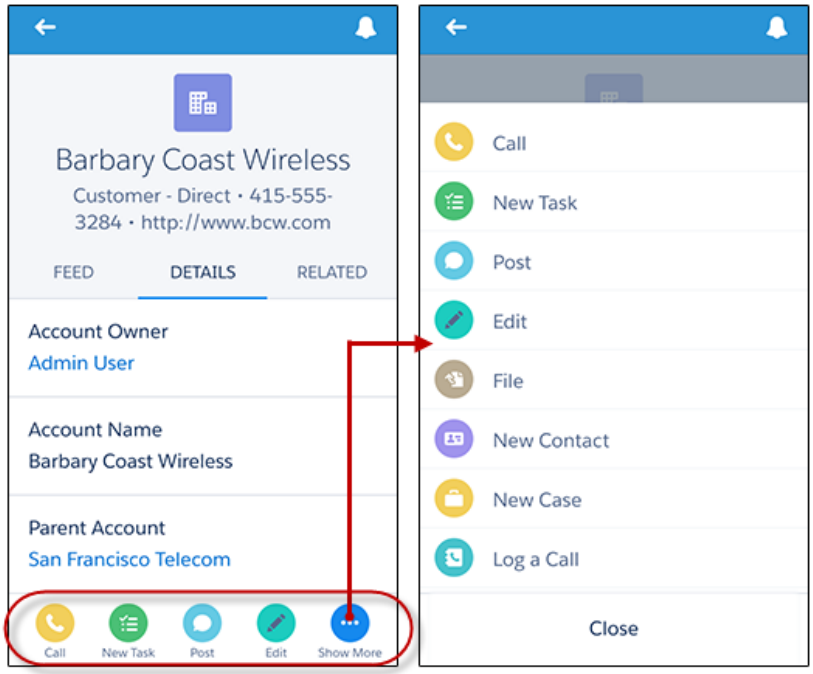
- Add CSS styling to a page that improves the usability of forms in a mobile user interface.

Implement Global Actions with Visualforce Pages

Actions enable users to do more in Salesforce and in Salesforce1. For example, you can let users create or update records and log calls directly in their Chatter feed or from their mobile device.

Actions, which appear in the action bar and action menu in Salesforce1, offer a quick way for mobile users to access commonly used tasks and functionality. Tapping the More icon ( ) opens the action menu, which contains the complete list of available actions.

Salesforce1 Actions in the Action Bar and Action Menu



Actions can be global or specific to an object.

- Global actions don't operate in the context of a specific object.
- Object-specific actions operate on a specific record.

Like navigation menu items, global actions don't operate in the context of a specific record. But unlike navigation menu items, global actions are typically used to implement micro-tasks. Generally, use navigation items to implement more complex processes or whole apps, and use global actions to implement micro-tasks. For example, a quick entry form where you fill in a few fields and save to a new record.

You can create both custom global and object-specific actions using Visualforce. This unit focuses on global actions, while the next unit covers object-specific actions.

Create a Visualforce Page for the Global Action

Global actions open a Visualforce page, which can in turn invoke additional pages. You need to define the first page before you create the action that launches it.

Let's create a global action that allows users to quickly create an account by capturing only the account name.

Note

As before, this example is simple for the purposes of clarity. We hope you find it informative, and extend it as you build your own Salesforce1 apps.

1. Open the Developer Console and click . Enter `QuickAccount` for the page name.
2. In the editor, replace any markup with the following.

```
<apex:page docType="html-5.0" title="Create Account">

    <apex:remoteObjects >
        <apex:remoteObjectModel name="Account" fields="Id,Name"/>
    </apex:remoteObjects>

    <div class="mypage">
        Account Name:
        <input type="text" id="accountName"/>
        <button onclick="createAccount()">Create Account</button>
    </div>

    <script>
        function createAccount() {
            var accountName =
document.getElementById("accountName").value;
            var account = new SObjectModel.Account();
            account.create({Name: accountName}, function(error, records)
{
                if (error) {
                    alert(error.message);
                } else {
                   sforce.one.navigateToSObject(records[0]);
                }
            });
        }
    </script>

</apex:page>
```

As before, if you click the **Preview** button to see your page, you'll see it in the standard desktop view, not the Salesforce1 version. As with our Latest Accounts page, you have to add it to Salesforce1 before you can see it inside Salesforce1. We'll get to that in a minute.

Also like Latest Accounts, Quick Account doesn't use a standard controller, and instead relies on Remote Objects for its behavior. Where Recent Accounts used Remote Objects just to load existing data, the Quick Account page uses Remote Objects to actually *create* a record in Salesforce. Remote Objects can also be used to update and even delete data where appropriate. It's an extremely versatile tool that's well-suited for use in Salesforce1 apps.

Beyond the Basics

While we're not learning about Remote Objects specifically, it's worth noting how Remote Objects uses `sforce.one` to navigate to the Account details page upon successful creation of the account. The magic happens in this call to `create`.

```
account.create({Name: accountName}, function(error, records)
{
    if (error) {
        alert(error.message);
    } else {
       sforce.one.navigateToSObject(records[0]);
    }
});
```

The `create` function accepts two arguments. The first is probably fairly obvious: it's a block containing name-value pairs that provide values for fields for the new account. If our Quick Account form had more fields, you could simply add appropriate name-value pairs to this block to create the account record with the additional field values. The second argument is a callback function. This function is called with the results of the Remote Objects operation once it completes.

Callback functions are a standard technique in JavaScript for handling events and asynchronous operations. Remote Objects uses this pattern to handle the response of its asynchronous operations. When you invoke a Remote Objects operation, you provide the parameters of the operation and, optionally, a callback function. Your JavaScript code continues uninterrupted after you invoke the operation. When the remote operation is completed and results are returned, your callback function is invoked and receives the results of the operation.

Here our callback function is extremely simple. It receives two values when called, an error object, and an array containing the records that were affected by the operation. If there's an error, the callback displays it in an alert. (That's not ideal for production use, but it works while we're exploring.) If the record was created successfully, we use the `sforce.one.navigateToSObject` function to navigate to the first item of the array of affected records. This just so happens to be the account that was created. Neat!

Create a Global Action with the Visualforce Page

Global actions aren't associated with any specific object, so they're created in their own section in Setup.

To use the QuickAccount page as a global action:

1. From Setup, enter `Visualforce Pages` in the Quick Find box, then select **Visualforce Pages**, and then enable the page for mobile apps.

   You learned how to enable a page for mobile apps in a previous unit.

2. Define the global action.

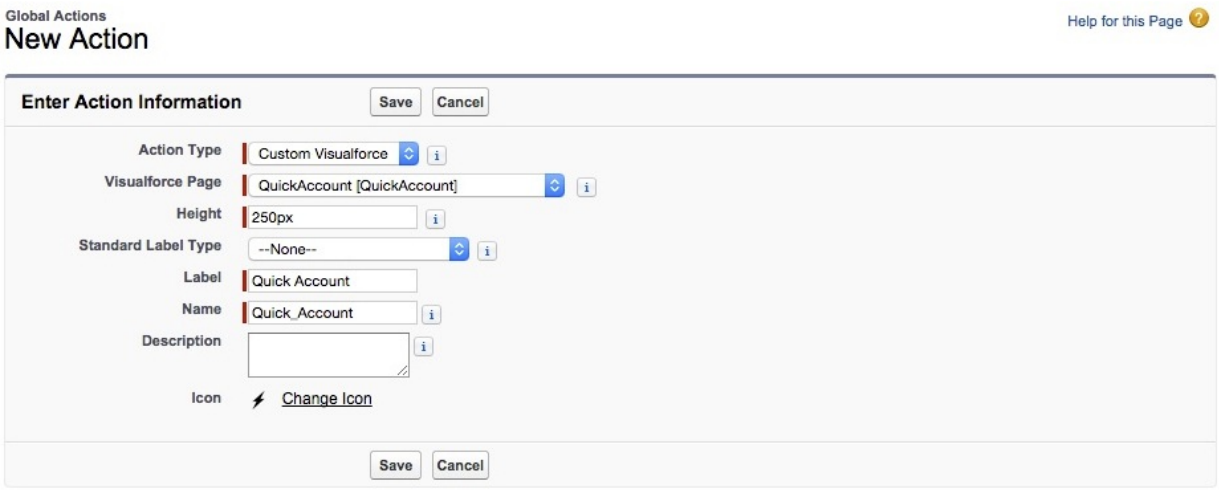
   a. From Setup, enter `Actions` in the Quick Find box, then select **Global Actions**.

   b. Click **New Action**.

   c. In the Action Type drop-down list, select Custom Visualforce.

   d. In the Visualforce Page drop-down list, select QuickAccount.

   e. In the Label field, enter `Quick Account`.
   The other default values are fine.



   f. Click **Save**.

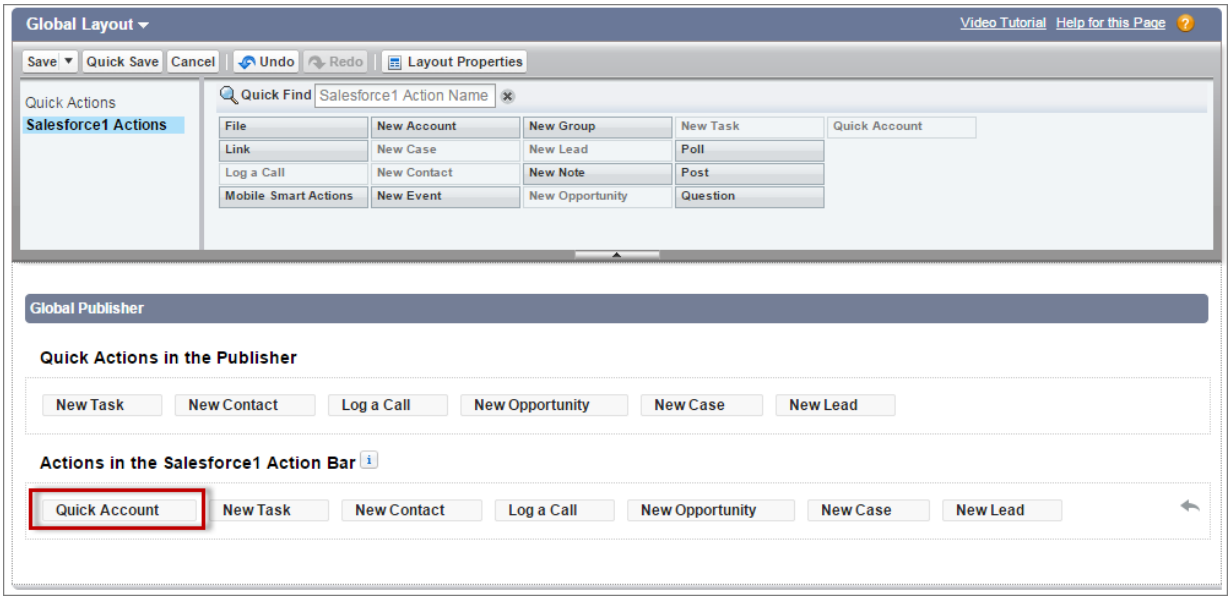
3. Add the global action to the publisher layout.

   a. From Setup, enter `Publisher Layouts` in the Quick Find box, then select **Publisher Layouts**.

   b. Click **Edit** to the left of **Global Layout**.

   c. In the Salesforce1 and Lightning Experience Actions, click to **override the predefined actions**.

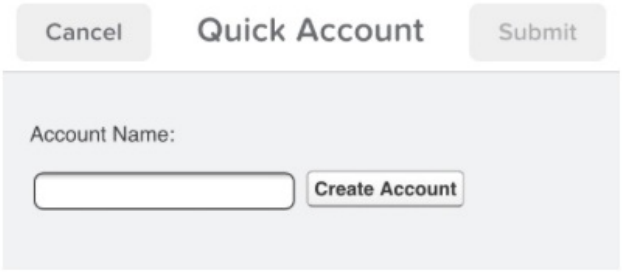
   d. From the Salesforce1 & Lightning Actions category in the palette, drag the Quick Account action to the Salesforce1 and Lightning Experience Actions section. Put it in the first position.



e. Click **Save**.

You're ready to test the new action. In a prior unit, we led you through the individual steps one-by-one. From here, we'll go more quickly.

To test your new global action, force a reload of your Salesforce1 app. Then, anywhere in Salesforce1, tap to access the action menu, and tap the **Quick Account** action. You should see your page presented, ready to create an account.



Give it a whirl! You should be able to create an account and then go to that account's detail page in Salesforce1. Overall, pretty good for a quick effort.

Style the Action Page for Mobile Devices

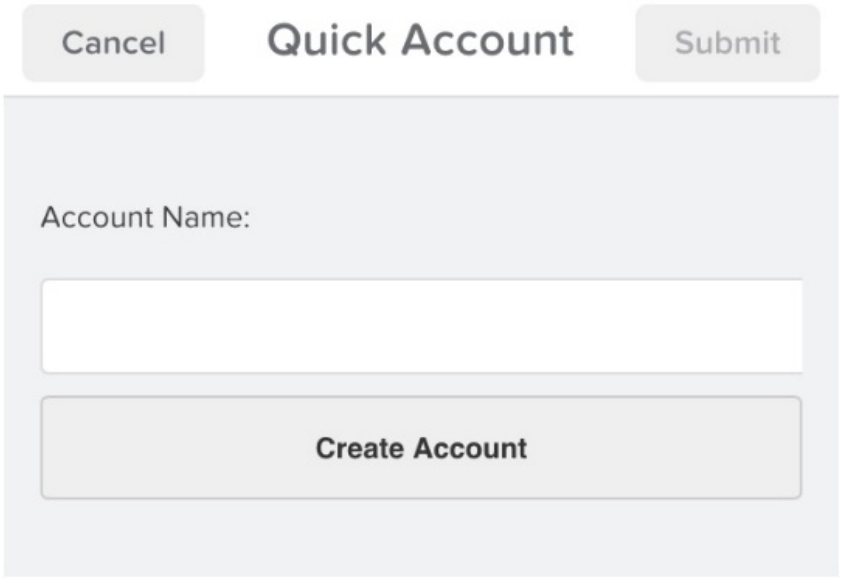
Use CSS to style the action's Visualforce page. Styling the page optimizes it for mobile devices and matches Salesforce1.

Once again it's time to optimize the look and feel of the page.

1. Open the Developer Console and click to open your page.
2. Below the opening tag in your page, add the following code.

```
<style>
    .mypage {
        font-family: "ProximaNovaSoft-Regular", Calibri;
        font-size: 110%;
        padding-top: 12px;
        width: 100%;
    }
    .mypage input[type=text] {
        width: 100%;
        height: 35px;
        -webkit-appearance: none;
        padding: 0 8px;
        margin: 4px 0;
        line-height: 21px;
        background-color: #fff;
        border: 1px solid #ddd;
        border-radius: 3px;
        outline: none;
    }
    .mypage button {
        -webkit-appearance: button;
        border-radius: 3px;
        display: block;
        padding: 12px;
        margin: 4px 0;
        width: 100%;
        background: #eee;
        border: solid 1px #ccc;
    }
</style>
```

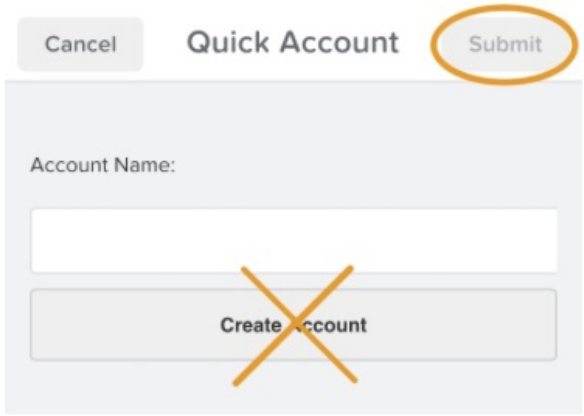
3. Reload the app and view your changes.



As before, text, entry fields, and tap targets are larger and more suitable for touch-based devices.

Improve Navigation with the Publisher SDK

Use the default **Submit** button that's part of the publisher dialog, instead of your own button, to provide a consistent user interface.



Global actions appear within the publisher, which gives them access to extra capabilities. The Publisher SDK provides a publish/subscribe API that allows you to send messages to the publisher and subscribe to publisher events. For example, you can send a `publisher.close` message to the publisher to close the publisher dialog. You can also subscribe to the `publisher.showPanel` and `publisher.post` events to be notified when the publisher dialog is displayed and when the submit button is clicked respectively.

Use the Publisher SDK to connect your custom features with standard publisher behavior and blend seamlessly with built-in Salesforce1 features. For example, to use the publisher submit button instead of your custom Create Account button in the QuickAccount page:

1. Open the Developer Console and click to open your page.
2. Above the existing block, add the following code to your page.

   ```
   <script src='/canvas/sdk/js/publisher.js'>
   </script>
   ```

   This code imports the Publisher SDK into your page.

3. Delete the for the Create Account button.
4. Replace the `createAccount()` function with the following code.

```
// When the panel is displayed, enable the submit button
Sfdc.canvas.publisher.subscribe({name: "publisher.showPanel", onData:function(e)
{
    Sfdc.canvas.publisher.publish(
        {name: "publisher.setValidForSubmit", payload: "true"});
}});

// When the submit button is pressed, create an account, and close the panel
Sfdc.canvas.publisher.subscribe({ name: "publisher.post", onData: function(e) {
    // Create the account using the Remote Object
    var accountName = document.getElementById("accountName").value;
    var account = new SObjectModel.Account();
    account.create({Name: accountName}, function(error, records) {
        if (error) {
            alert(error.message);
        } else {
            //  Close the publisher panel
            Sfdc.canvas.publisher.publish(
                { name: "publisher.close", payload:{refresh:"true"}});
        }
    });
}});
```

When you reactivate the global action, your action panel uses the built-in Salesforce1 navigation elements, giving your users a consistent experience across the entire Salesforce1 app.

Beyond the Basics

The Publisher SDK is part of a larger toolkit, the Force.com Canvas SDK. The Force.com Canvas SDK is a suite of JavaScript libraries that provide simple ways to use existing Salesforce1 APIs (REST API, SOAP API, Chatter REST API) to build better user experiences for your mobile users. The Force.com Canvas SDK gives you capabilities that go beyond the simple navigation we're using here. When you're ready to take the next step in your mobile apps, be sure to check it out.

Implement Object-Specific Actions with Visualforce Pages

Learning Objectives

After completing this unit, you'll be able to:

- Describe the two most significant requirements of an object action, as compared to a global action.
- Implement a Salesforce1 object action using a Visualforce page.
- Use the Publisher SDK to close the publisher and refresh the object page.

Implement Object-Specific Actions with Visualforce Pages

When you create an object-specific action using a Visualforce page, that page is executed with the current object. In other words, a specific record ID is associated with the Visualforce page.

Visualforce pages used to implement object-specific actions must use the standard controller for that object. The standard controller automatically loads the record for the record ID.

Generally, when the action completes, users are redirected to a page related to the originating record.

Create a Visualforce Page for the Object-Specific Action

Object-specific actions open a Visualforce page, which can in turn invoke more pages. You must define the first page before you can create the action that launches it.

As an example, let's create an object-specific action that provides a simple two-button widget to close an opportunity as either won or lost.

1. Open the Developer Console and click . Enter `CloseOpportunity` for the page name.
2. In the editor, replace any markup with the following.

```
<apex:page docType="html-5.0" standardController="Opportunity" title="Close Opportunity">

    <script src='/canvas/sdk/js/publisher.js'></script>

    <apex:remoteObjects>
        <apex:remoteObjectModel name="Opportunity" fields="Id,Name"/>
    </apex:remoteObjects>

    <div class="mypage">
        <button onclick="closeOpportunity('Closed Won')">Won</button>
        <button onclick="closeOpportunity('Closed Lost')">Lost</button>
    </div>

    <script>
        var opportunityId = "{!Opportunity.Id}";

        function closeOpportunity(stageName) {
            var opportunity = new SObjectModel.Opportunity();
            opportunity.update([opportunityId], {StageName: stageName}, function(error, records) {
                if (error) {
                    alert(error.message);
                } else {
                    Sfdc.canvas.publisher.publish({ name: "publisher.close", payload:
{refresh:"true"}});
                }
            });
    }
    </script>

</apex:page>
```

This page couldn't be more simple, but it illustrates a number of details about writing your own object actions. First, the page uses the standard controller for opportunities, as required. But notice something interesting: it only uses the standard controller to *load* the details for the opportunity. Once again, the page uses Remote Objects, this time to update an existing record. So the page loads data using the standard controller, but updates it with Remote Objects, combining two different data access methods on the same page.

Beyond the Basics

Why not use the standard controller to update the record, too? The simple answer is, Remote Objects fits better with the page interaction model used by Salesforce1. You *can* use standard Visualforce in Salesforce1, but to optimize your pages for the mobile environment, you're better off using Visualforce tools like Remote Objects or JavaScript remoting. They connect better with JavaScript-based Salesforce1 features like `sforce.one` and the Publisher SDK. They also perform fewer requests with smaller data payloads, which can improve performance quite a bit for mobile devices.

Finally, we handle a successful update of the opportunity using a callback function similar to the one defined in the Quick Account page. Except this time, instead of using a `sforce.one` navigation function, we use the Publisher SDK to close the publisher and refresh the opportunity we were on.

Create an Object-Specific Action with the Visualforce Page

Object-specific actions are associated with a particular object, so they're created on the object in Setup.

To create an action for Opportunities with the CloseOpportunity page, do the following.

1. From Setup, enter `Visualforce Pages` in the Quick Find box, then select **Visualforce Pages**, and then enable the page for mobile apps.

   You learned how to enable a page for mobile apps in a previous unit.

2. Define the action for the object.

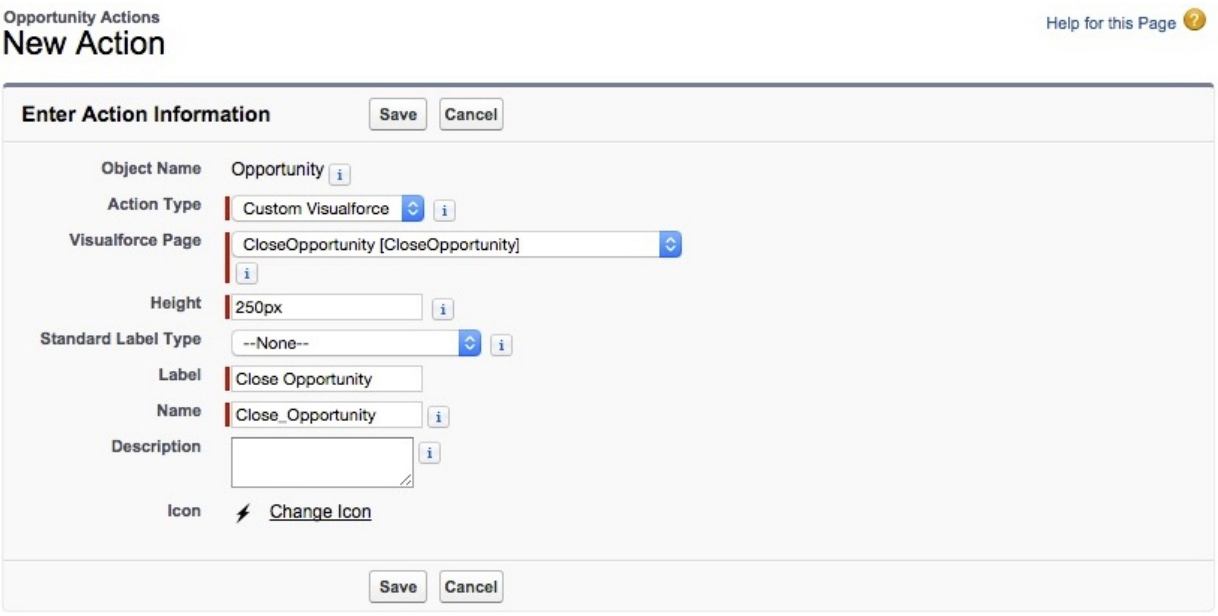
   a. From Setup, click .

   b. Click **New Action** .

   c. In the Action Type drop-down list, select Custom Visualforce.

   d. In the Visualforce Page drop-down list, select CloseOpportunity.

   e. In the Label field, enter `Close Opportunity`.
   The other default values are fine.



f. Click **Save**.

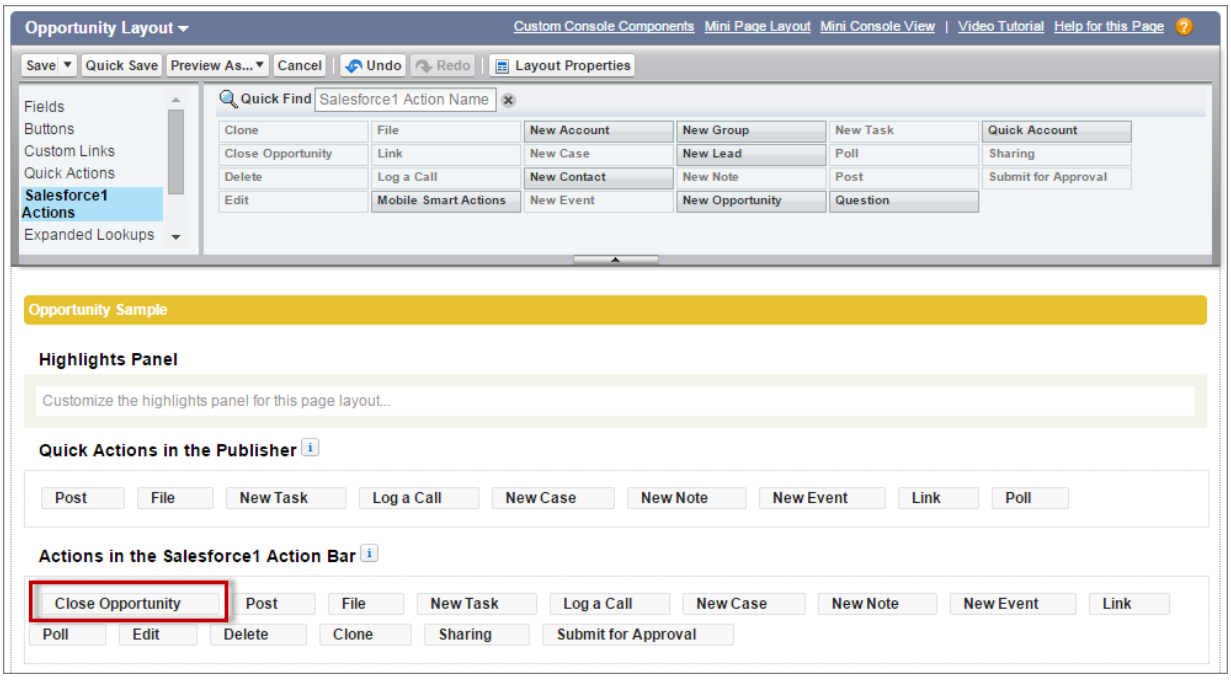
3. Add the action to the publisher layout.

   a. From Setup, click .

   b. Click the **Edit** link to the left of Opportunity Layout.

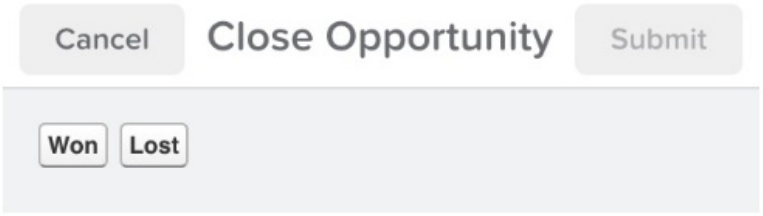
   c. In the Salesforce1 and Lightning Experience Actions, click to **override the predefined actions**.

   d. Click **Salesforce1 & Lightning Actions** in the palette and then drag the Close Opportunity action to the Salesforce1 and Lightning Experience Actions section.



   e. Click **Save**.

To test your new global action, force a reload of your Salesforce1 app. Then navigate to an opportunity record, tap to access the action menu, and tap the **Close Opportunity** action. You should see your page, ready to close the opportunity.



Naturally, you're a sales rock star, so go ahead and close that opportunity as won. But before we start spending that commission check, there's some additional work to do.

Style the Action Page for Mobile Devices

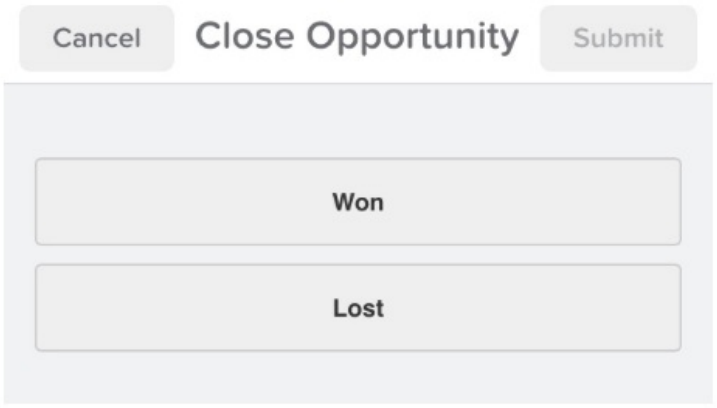
Use CSS to style the action's Visualforce page to optimize it for mobile devices, and to match Salesforce1.

Once again it's time to optimize the look and feel of the page.

1. Open the Developer Console and click to open your page.
2. Below the opening tag, add the following code to your page.

```
<style>
    .mypage {
        font-family: "ProximaNovaSoft-Regular", Calibri;
        font-size: 110%;
        padding-top: 12px;
    }
    .mypage button {
        -webkit-appearance: button;
        border-radius: 3px;
        display: block;
        padding: 12px;
        margin: 8px 0;
        width: 100%;
        background: #eee;
        border: solid 1px #ccc;
      }
</style>
```

3. Reload the app and view your changes.



We might sound like a broken record by now, but the styling changes you've implemented significantly improve the usability of the action you're creating, with little effort. Great work!

Use Visualforce Pages in Page Layouts and Mobile Cards

Learning Objectives

After completing this unit, you'll be able to:

- Describe two differences between adding a Visualforce page to a page layout section or a mobile card.
- Create Visualforce pages to use in Salesforce1 page layouts.
- Create Visualforce pages for mobile cards in Salesforce1.

Use Visualforce Pages in Page Layouts and Mobile Cards

Customize record detail pages by embedding Visualforce pages in your page layouts, or by adding them as mobile cards on a page layout.

Customizing page layouts by embedding Visualforce in a section on the page has long been a feature of Salesforce. This feature works as well with Salesforce1 as it does with regular Salesforce. Pages added to a layout this way appear in both the standard desktop version of Salesforce and in Salesforce1.

Mobile cards, however, allow you to add a Visualforce page to a layout for mobile users only. Mobile cards are a great way to add small pieces of information and to format them specifically for mobile use.

A Visualforce page used in a page layout or as a mobile card runs in the context of a specific record. As with object-specific actions, a specific record ID is associated with the page and it must use the standard controller for that object.

Create a Visualforce Page to Use on Page Layouts

Design embeddable Visualforce pages by using the standard controller and keep the page focused, ideally on a single task or kind of information.

As an example, let's create a Visualforce page that provides a stock quote for the selected account.

1. Open the Developer Console and click . Enter `StockQuote` for the page name.
2. In the editor, replace any markup with the following.

```
<apex:page docType="html-5.0"
standardController="Account">
    <style>
        .mypage .quote {
            margin: 12px 0;
            font-size: 64px;
            text-align: center;
        }
        .mypage .delta {
            font-size: 24px;
            text-align: center;
            color: green;
        }
    </style>

    <div class="mypage">
        Stock:
        <div class="quote">$42.00</div>
        <div class="delta">+1.32%</div>
    </div>

</apex:page>
```

Well, this page isn't hard to understand! Indeed, this page is almost entirely static, simple HTML. Note the use of the standard controller for Account, which lets you add this page on an Account layout as an embedded page or mobile card.

What's important here isn't any specific programmatic technique, so much as understanding the best uses for embedded pages and mobile cards. Especially in a mobile environment, keep your Visualforce focused on the essential information a user needs to accomplish a specific task. Simple mobile apps perform better, and users can stay focused on the job they're doing.

Keep in mind the context where your app will be used—in the car, while walking into an appointment, while waiting in line, and so on. You might be tempted to add extra fields or information, "just in case." Don't. *Focus*. If users need details, they can zoom into them on another page.

Use the Page on a Page Layout or Mobile Card

Add a Visualforce page to a section on a page layout to add it for all users. Add it as a mobile card on a page layout to add it only to Salesforce1.

The steps are similar whether you're adding the page as a section on a page layout or as a mobile card.

1. From Setup, enter `Visualforce Pages` in the Quick Find box, then select **Visualforce Pages**, and then enable the page for mobile apps.

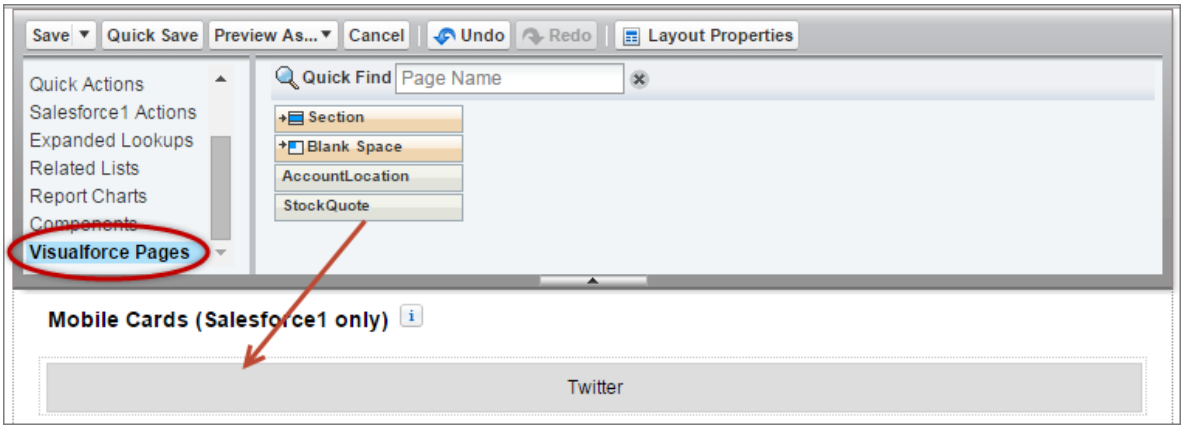
   You learned how to enable a page for mobile apps in a previous unit.

2. Add the Visualforce page to a page layout.

   a. From Setup, click .

   b. Click the **Edit** link to the left of Account Layout.

c. Select **Visualforce Pages** and drag the **StockQuote** page to where you want it displayed on the page.



Pages added to the **Mobile Cards** section only appear in Salesforce1. Pages added to another section on the page layout are visible to both mobile and desktop clients.

d. Click **Save**.

To test your new mobile card, force a reload of your Salesforce1 app. Then navigate to an account record and swipe to the Related panel. You should see something like the following.

Although this stock widget is hard-coded, you could easily connect it to a stock quotation lookup service you write in Apex, or a Web Services callout, or any of various other data sources. You can add multiple mobile cards to a layout, and they'll all appear in the Related panel.

Resources

Adopt User Interface Guidelines

Learning Objectives

After completing this unit, you'll be able to:

- List at least six user interface guidelines for building Visualforce pages running in Salesforce1.
- Explain two characteristics of responsive design and its importance in mobile development.
- Build a simple responsive user interface using CSS.
- List at least six Visualforce components you shouldn't use when building Visualforce pages for Salesforce1.

User Interface Guidelines

Visualforce pages running in Salesforce1 aren't automatically optimized for mobile devices. As the developer, you're responsible for implementing the appropriate best practices to deliver a great user experience to your mobile users.

Here are some guiding principles for building Visualforce pages running in Salesforce1.

- Design for small screens. On a mobile device, less is more. Limit the amount of data you show or capture to what is strictly necessary. Mobile pages don't have to support all the bells and whistles.
- Use responsive design to automatically adapt the page layout to different screen sizes.
- Design for touch interactions. Avoid keyboard or mouse-centric user interface components and metaphors. Provide large tap areas.
- Limit keyboard input.
- Use available device sensors—for example, geolocation and camera—when applicable.
- Avoid Visualforce components that mimic the full Salesforce site. In general, prefer plain HTML components.
- Prefer the JavaScript single-page application pattern over multi-page processes. A single-page application is an app that fits in a single page and provides a more fluid user experience akin to a native app. Views are created dynamically in JavaScript and injected into the DOM as users navigate through the app's features.
- Use a JavaScript framework. Building single-page applications is challenging, and existing JavaScript frameworks handle most of that complexity.
- Use a CSS framework. Styling HTML components for an optimized mobile experience isn't trivial, and existing user interface frameworks can help a lot.

Match the Salesforce1 Visual Design

Visualforce pages running in Salesforce1 by default have minimal styling and don't match the Salesforce1 visual design.

Create Visualforce pages that more closely match the look of Salesforce1 by using CSS styling and minimal HTML.

Fonts and Styles

The visual design for Salesforce1 uses Proxima Nova, a sans-serif typeface licensed specially for Salesforce1. People in your organization might not have the font installed on their devices, but the following CSS font stack will use the closest matching font available:

```
p {
    font-family: "ProximaNovaSoft-Regular", Calibri,
        "Gill Sans", "Gill Sans MT", Candara, Segoe, "Segoe
UI",
         Arial, sans-serif;
}
```

Change the selectors to match the appropriate text blocks and styles on your pages.
We also recommend slightly increasing the size of the font used for normal text and form elements. This makes them easier to read and, for form elements, to tap into. Here's a complete style block to get you started:

```
<style>
    html, body, p {
        font-family: "ProximaNovaSoft-Regular", Calibri,
            "Gill Sans", "Gill Sans MT", Candara,
Segoe,
            "Segoe UI", Arial, sans-serif;
        font-size: 110%;
    }
    input { font-size: 95%; }
</style>
```

Warning

Directly referencing Salesforce1 style sheets in your pages, or depending on styles found in them, isn't supported. As Salesforce1 evolves, the styles will change in ways that you won't expect. Pages that depend on unsupported styles might eventually break.

HTML Markup

If your organization's Visualforce pages match the full Salesforce site's look and feel, it's probably by using the Salesforce header, sidebar, styling, and and child components. Don't explicitly disable these elements, by setting ; Salesforce1 automatically disables these elements when the page runs inside it.

However, you might want to reconsider using and child components, because they explicitly match the full Salesforce site look and feel, and even with standard stylesheets turned off, the markup they generate carries over into Salesforce1. This makes your markup more complex, and it's harder to apply styling that matches Salesforce1.

Instead, use simple, static HTML markup, or the markup required by your chosen Mobile Toolkit. Sample markup is included the next section.

Style Guides and Toolkits

These practices alone will make your pages look somewhat like Salesforce1, but won't fully match the app's look-and-feel. For more detailed style matching guidelines, see the Salesforce1 Style Guide. You can also adopt one of several toolkits available that attempt to provide complete solutions to matching Salesforce1's visual design. See this unit's resources for the names of and links to a number of good options.

Use Responsive Design

Responsive design is a Web-design method aimed at creating Web user interfaces that provide an optimal viewing experience, including easy reading and navigation, on various screen sizes.

A responsive user interface adapts the layout to the screen size by using fluid, proportion-based grids, flexible images, and CSS3 media queries. Using responsive design, you can create Visualforce pages that look and work great on phones and tablets.

Standard Salesforce1 pages use responsive design techniques to provide device-optimized layouts. The primary technique used is a stacked single-column layout for phones, and a side-by-side, two-column layout for tablets. The page is the same for all devices, and adapts to the actual screen size it's displayed on.

You can use a similar technique for your Visualforce pages in Salesforce1.

Let's examine a simple responsive layout implementation.

1. Open the Developer Console and click . Enter `ResponsivePage` for the page name.
2. In the editor, replace any markup with the following.

```
<style>
/* Default CSS: 2 columns */
.content-block {
    width: 50%;
    float: left;
}

<!-- add more styles here --
>

</style>
```

   This code establishes the default layout, which places two equal-sized content blocks—for example, two

   elements—side-by-side on the page.

3. In the editor, above , add the following to the block.

```
/* CSS phone */
@media screen and (max-width: 767px)
{
    .content-block {
        width: 100%;
        float: none;
    }
}
```

The CSS styling associated with the media query and size restriction overrides the default style when the screen size is small (phone-sized). It also changes the width of the content blocks to 100%. When viewed on a phone the content blocks stack on top of each other, rather than sit side-by-side.

4. In the editor, below the block of the previous code, add the following.

```
<!-- HTML for two blocks of content
     On a phone they position one on top of the other
     On larger screens they position next to each other -->

<div class="content-block">
    This block is on top on the phone, and on the left on larger screens.
</div>

<div class="content-block">
    This block is on the bottom on the phone, but on the right on larger
screens.
</div>
```

View the Visualforce page in a browser, and then drag the corners of your browser window to observe the responsive layout.

Visualforce Components and Features to Avoid in Salesforce1

Most core Visualforce components (those components in the `apex` namespace) function normally within Salesforce1. Unfortunately, that doesn't mean they're optimized for mobile, or that every feature works with Salesforce1. You can improve the Salesforce1 user experience of your Visualforce pages by following some straightforward rules.

In general, avoid structural components, like and child components, and other components that mimic the Salesforce look and feel, such as . If you must use these components, set them to one column, using , instead of the default of two columns.

Avoid wide, non-wrapping components, especially , , , and , which are all unsupported. Keep device width in mind when creating tables with .

Avoid using . Inline editing is a user interface pattern that works well for mouse-based desktop apps, but it's difficult to use on a touch-based device, especially on phones where the screen is small.

Using is fine for fields that display as a basic input field, like text, email, and phone numbers, but avoid using it for field types that use an input widget, such as date and lookup fields.

PDF rendering, by setting `renderAs="PDF"` on , isn't supported for pages in Salesforce1.

Use CSS and JavaScript Mobile Frameworks

Learning Objectives

After completing this unit, you'll be able to:

- List at least three CSS frameworks you can use to style your Visualforce pages.
- List at least three JavaScript frameworks you can use to build Visualforce pages using the single page application pattern.
- Describe two technical challenges frameworks can simplify in your Salesforce1 development efforts.

Using Mobile Frameworks

A mobile Web framework makes it easy to support the single-page application pattern when writing Visualforce pages for Salesforce1.

Getting apps to render perfectly on different screen sizes can be tedious. Styling every user interface component to provide an optimized mobile experience is hard as well. Use existing CSS frameworks that have already done the hard work for you. To build mobile-optimized user interfaces you can use, for example, the following user interface frameworks.
Writing single-page apps from scratch is complex. Fortunately, you can use existing JavaScript frameworks that abstract and manage most of that complexity. To build Visualforce pages using the single-page application architecture, you can use, for example, the following JavaScript frameworks.

These frameworks are just a few of the options available. Take a look at a few different options and decide which one matches the skills and preferences of your organization. To get you started, we'll take a brief look at a couple of them. We don't provide a recommendation—we're simply showing you the first few steps along a path. You're in charge of your journey! Use one of the options presented here, or blaze your own trail. The principles you learn here will prove useful along the way.

Bootstrap

Bootstrap is a popular user interface framework for developing responsive Web apps.

Using the Bootstrap responsive grid, it's easy to build Visualforce pages with a responsive layout, that is, Visualforce pages with a layout that adapts to different screen sizes.

Watch this video for an introduction on using Bootstrap in Visualforce pages.

Watch this video for an introduction to building Salesforce1 apps with Visualforce, AngularJS, and Bootstrap.

Ionic

Ionic is a user interface library that provides mobile-optimized UI components to build native-feeling mobile Web apps.

Unlike Bootstrap and Ratchet, Ionic isn't agnostic in terms of the underlying architectural framework you use: Ionic is built on top of AngularJS to provide a complete solution for developing apps that are both well architected and native-feeling.

Watch this video for an introduction to building Salesforce1 apps with Ionic and AngularJS.

Sample Force.com Mobile Application with Ionic and AngularJS demonstrates how to build a Salesforce mobile app with the Ionic framework and the Salesforce Mobile SDK.

Build a Salesforce1 App with the Ionic Mobile Framework

For the rest of this unit, we'll explore using Ionic and AngularJS to create another simple Salesforce1 app to display a list of contacts.

As in other units, this app is simple, but offers many opportunities for you to extend on your own. We can't provide a complete tutorial for learning Ionic and AngularJS, but this lesson will give you a feel for developing apps using a mobile framework.

Create the App's Visualforce Page

This page is a container for the app's resources and functionality.

This Visualforce page defines the app's URL and provides a "shell" in which the app runs.

1. Open the Developer Console and click . Enter `IonicContacts` for the page name.
2. In the editor, replace any markup with the following.

```
<apex:page showHeader="false" sidebar="false" standardStylesheets="false"
    docType="html-5.0" applyHtmlTag="false" applyBodyTag="false">

<html>

<head>
    <link href="https://code.ionicframework.com/1.0.0-beta.14/css/ionic.css"
rel="stylesheet"/>
    <script src="https://code.ionicframework.com/1.0.0-beta.14/js/ionic.bundle.js"></script>
    <!-- script src="{!URLFOR($Resource.IonicContactsApp)}"></script -->
</head>

<body ng-app="starter">

    <apex:remoteObjects>
        <apex:remoteObjectModel name="Contact" fields="Id, Name, Title"/>
    </apex:remoteObjects>

    <ion-nav-bar class="bar-positive">
        <ion-nav-back-button/>
    </ion-nav-bar>

    <ion-nav-view/>

</body>

</html>

</apex:page>
```

At first glance, this code looks similar to pages created in other units in this module. But when you look at it more closely, you'll see some interesting differences. The only thing that *isn't* new is the Remote Objects definition, in this case, to work with contacts.

First, the tag has a new set of attributes. This combination of attributes, all set to `false`, is something you normally use when building single-page apps. This combination is the way to (mostly) control the HTML that's generated by Visualforce. Visualforce sets the document type to HTML5 (`docType="html-5.0"`), and doesn't add its own , , or tags to the page (`applyHtmlTag="false"` and `applyBodyTag="false"`). Combine these with the familiar attributes to suppress the standard Salesforce header, sidebar, and stylesheets to set your Visualforce for modern, full-screen HTML development.

The flip side of taking control from Visualforce is that, with these settings, *you're* responsible for adding these elements to the page. So, notice the static HTML tags for , , and .

Notice also the new attribute on the : `ng-app="starter"`. This attribute is an annotation that AngularJS uses to indicate the name of the app to run wher the page is loaded. This is how an AngularJS app gets launched. You'll create the definition for this app in the next step. (The line referencing `$Resource.IonicContactsApp` is commented out because the definition isn't created yet. You'll fix that in the next step too.)

Finally, these not-quite-HTML tags are unique to the Ionic framework.

- is an Ionic user interface component that displays the application header and navigation.
- is an Ionic container for the views of your application. It serves as a placeholder for content that's generated as the app runs.

Note

JavaScript frameworks evolve rapidly. The URLs included here were current during writing. To ensure that you include the latest version of the Ionic stylesheet and JavaScript library, visit code.ionicframework.com.

Create the Application Resource

Add shared JavaScript code to your app using a static resource.

You can add JavaScript within tags on a Visualforce page. But if the code is shared across pages, it's better to put it in a static resource that can be cached across requests for better performance. When building single-page apps, you'll often want to isolate your JavaScript code into its own resource.

1. To create a static resource for the main app code, open the Developer Console and click . Enter `IonicContactsApp` as the name and `text/javascript` as the MIME type for the resource.

2. In the editor, replace the contents with the following.

```
angular.module('starter', ['ionic'])

    .config(function($stateProvider) {

        $stateProvider
            .state('contactlist', {
                url: '',
                templateUrl: '/apex/IonicContactsList_Tpl',
                controller : "ContactListCtrl"
            })

    })

    .controller('ContactListCtrl', function($scope) {

        var contactsRemote = new SObjectModel.Contact();

        contactsRemote.retrieve({ limit: 10 }, function(err, records, event){
            if (err) {
                alert(err.message);
            } else {
                var contacts = [];
                for (var i = 0; i < records.length; i++) {
                    var contact = records[i];
                    contacts.push({id: contact.get("Id"), name: contact.get("Name"), title:
contact.get("Title")});
                }
                $scope.$apply(function() {
                    $scope.contacts = contacts;
                });
            }
        });

    })
```

3. In the Developer Console, switch back to the IonicContacts page and remove the comment tags from the second tag, so that it reads as follows.

```
<script src="{!URLFOR($Resource.IonicContactsApp)}"></script>
```

This JavaScript file is the heart of the app. It defines an AngularJS module, “starter”, which is referenced in the element of the page created in the prior step. AngularJS launches using this code when the page is loaded.

So what is the code? An Ionic app can have different screens represented by states. You can think of each state as a different type of screen which represents “list,” “detail,” and “edit,” for example. Each state matches a specific URL fragment.

This specific app has a single state, named `contactlist`, which shows a list of contacts. The `contactlist` state definition specifies that the IonicContactsList_Tpl template (which we’ll create in the next step) provides the user interface. The ContactListCtrl controller provides the user interface logic.

The ContactListCtrl controller isn’t the usual Visualforce controller. Instead, it’s a client-side AngularJS controller, defined here in the JavaScript, in the block that begins with `.controller('ContactListCtrl' ...)`. This controller uses a Remote Objects call to retrieve a list of contacts. It then makes that list available in an AngularJS scope so that the IonicContactsList_Tpl template can render it.

This example code might not make complete sense until you’re more familiar with AngularJS and Ionic. And it might look complicated for such a simple list. The power of using Ionic (and AngularJS behind it) becomes apparent as you extend this example to add, for example, a contact details screen and search functionality.

Create the List Template

Create a template file for each state, or screen, in the app.

Here you only need to create one template, for the contacts list screen. The template is just another Visualforce page.

1. Open the Developer Console and click . Enter `IonicContactsList_Tpl` for the page name.
2. In the editor, replace any markup with the following.

```
<apex:page showHeader="false" sidebar="false"
standardStylesheets="false"
    applyHtmlTag="false" applyBodyTag="false">

<ion-view view-title="Contacts">

    <ion-content id="content" class="has-header">

        <ion-list>
            <ion-item ng-repeat="contact in contacts">
                {{ contact.name }}
                <p>{{ contact.title }}</p>
            </ion-item>
        </ion-list>

    </ion-content>

</ion-view>

</apex:page>
```

The template uses Ionic user interface directives like `ion-content` and `ion-list`. These directives are part of the standard complement of Ionic UI elements.

The template's real work happens in the `ion-list`. There, an AngularJS `ng-repeat` iterates through the list of contacts that the controller made available in the scope, in the app's JavaScript resource created in the previous step. For each contact, the contact's name and title are displayed, using the AngularJS data binding notation, for example, `{{ contact.name }}`. The `{{` double braces `}}` tell AngularJS to evaluate the contents.

By breaking each state's screen definition into its own template file, Ionic makes it easy to create a complex single-page app, while still working in individual screens when you're designing the user interface.

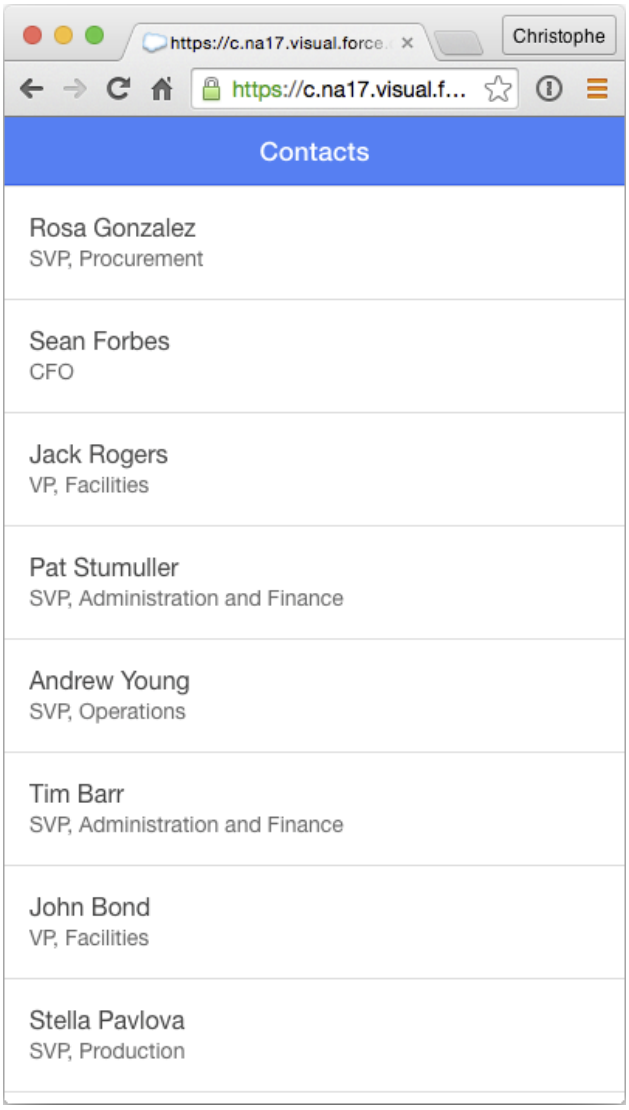
Test the App!

You can now test the app in a browser or in Salesforce1.

Because this app isn't using any Salesforce1-specific features, such as the `sforce.one` navigation utility or the Publisher SDK, you can try it right in your browser.

1. In the Developer Console, return to the IonicContacts page.
2. Click **Preview**.

You should see the contacts list in the browser.



Of course, there's nothing like testing it in Salesforce1 on an actual device. Where would you add this page to Salesforce1? How would you add it? Take five minutes right now and test your memory. Can you add this page to Salesforce1 without referring to a prior lesson?

Resources

Hands-on Challenge +500 points

Create an Account Using REST API and Workbench

Using REST API and Workbench, create an account with the name "Blackbeards Grog Emporium" and the description "The finest grog in the seven seas."

Log into Trailhead